APPENDIX - A

Tabula Rasa delivers solutions that use customer identification technologies to improve quality of service and enhance the way that Tabula Rasa clients interact with their customers. The Tabula Rasa Component Framework (TRCF) provides a component-based framework for rapidly deploying solutions customized to client's needs and capabilities. This document describes the system architecture design of the TRCF.

The TRCF is designed to be flexible enough to be deployed in any of Tabula Rasa's potential markets, such as the hospitality or entertainment industries. The design is not based on any particular industry or business, but has been normalized to provide a basic framework, which can be easily tailored to any business application. The TRCF is platform independent, secure, and fault-tolerant. Based on J2EE (Java 2 Enterprise Edition) architecture, the TRCF is flexible, scalable and extensible. On the client side, it is designed to make highly efficient use of remote device resources (e.g. RF ID readers, digital cameras, etc). On the server side, it has minimal impact on a customer's existing IT infrastructure and offers clean and simple integration with enterprise applications.

The TRCF provides the architectural framework and toolset needed to expedite the rapid development of Tabula Rasa solutions. Using the TCRF reduces time to market, lowers development costs, and provides a common platform for support and upgrades.

Introduction

Tabula Rasa is focused on developing solutions that use customer identification technologies to improve the quality of service and expand the types of services offered by Tabula Rasa clients. The Tabula Rasa Component Framework (TRCF) is intended for use as a rapid application development toolkit. It will enable Tabula Rasa to rapidly develop and deploy component based applications on a variety of server platforms.

The TRCF will not be targeted towards any one industry. Instead, it will be flexible enough to use for any of Tabula Rasa's potential markets, such as the hospitality or entertainment industries. Solutions built using the TRCF will be tailored to the specific business needs of Tabula Rasa clients.

Part of the challenge in building this kind of framework is defining the functionality required to support a customized system without specifying the end user requirements. In other words, a detailed understanding the specific applications that may be deployed for Tabula Rasa customers is outside the scope of this effort. Instead, this project is focused on the functionality that can be commonly used in a range of customized applications.

The TRCF must be platform independent, secure, and fault-tolerant. On the client side, it must make highly efficient use of remote device resources (e.g. RF ID readers, digital cameras, etc). On the server side, it must have minimal impact on a customer's existing IT infrastructure while offering clean and simple integration with enterprise applications.

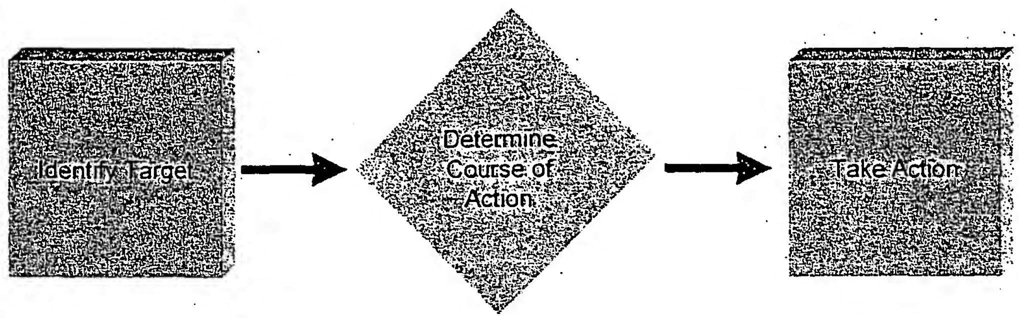
In short, the TRCF will provide the architectural framework and toolset needed to expedite the rapid development of Tabula Rasa solutions. Using the TCRF will help reduce time to market, lower development costs, and ensure high quality deliverables. The goal of this document is to describe the functionality encapsulated in the TRCF.

Approach

The challenge in defining the TRCF was to insulate the design from detailed user requirements so as not to bias the resultant architecture. While a multitude of business examples were discussed and used as reality checks against what was being proposed, no detailed user requirements are included in this document. While the team initially sought to undertake traditional business analysis using the Use Case modeling technique from the Unified Modeling Language (UML), this approach was abandoned after the diagrams were completed, as it became apparent that any specific business scenario target would slant the framework. The team moved on from the use case diagrams to develop the component functions and logical component diagram. The use case diagrams, component functions and logical component diagram are intended to represent generic interaction with the "core" or framework of the TRCF service/product offering.

System Workflow

The basic sequence of events in the system workflow is illustrated in Figure 1 below. These steps define the Tabula Rasa product/service offering, and all components of the TRCF should map to one or more of these steps.



Basic TRCF workflow

Fig. 1

Step	Description
Identify Target	The system senses a target and associates it with an identifier.
Determine Course of Action	Based upon the information gathered in the "Identify Target" step, the system determines and triggers an appropriate action command or series of action commands.
Take Action	The system, or an operative, executes the action(s).

Use Case Diagrams

The following sections describe the system in terms of the use case diagrams. Use case diagrams show interaction between users of the system, or actors, and the key function of the system, or use cases.

Actor Definitions

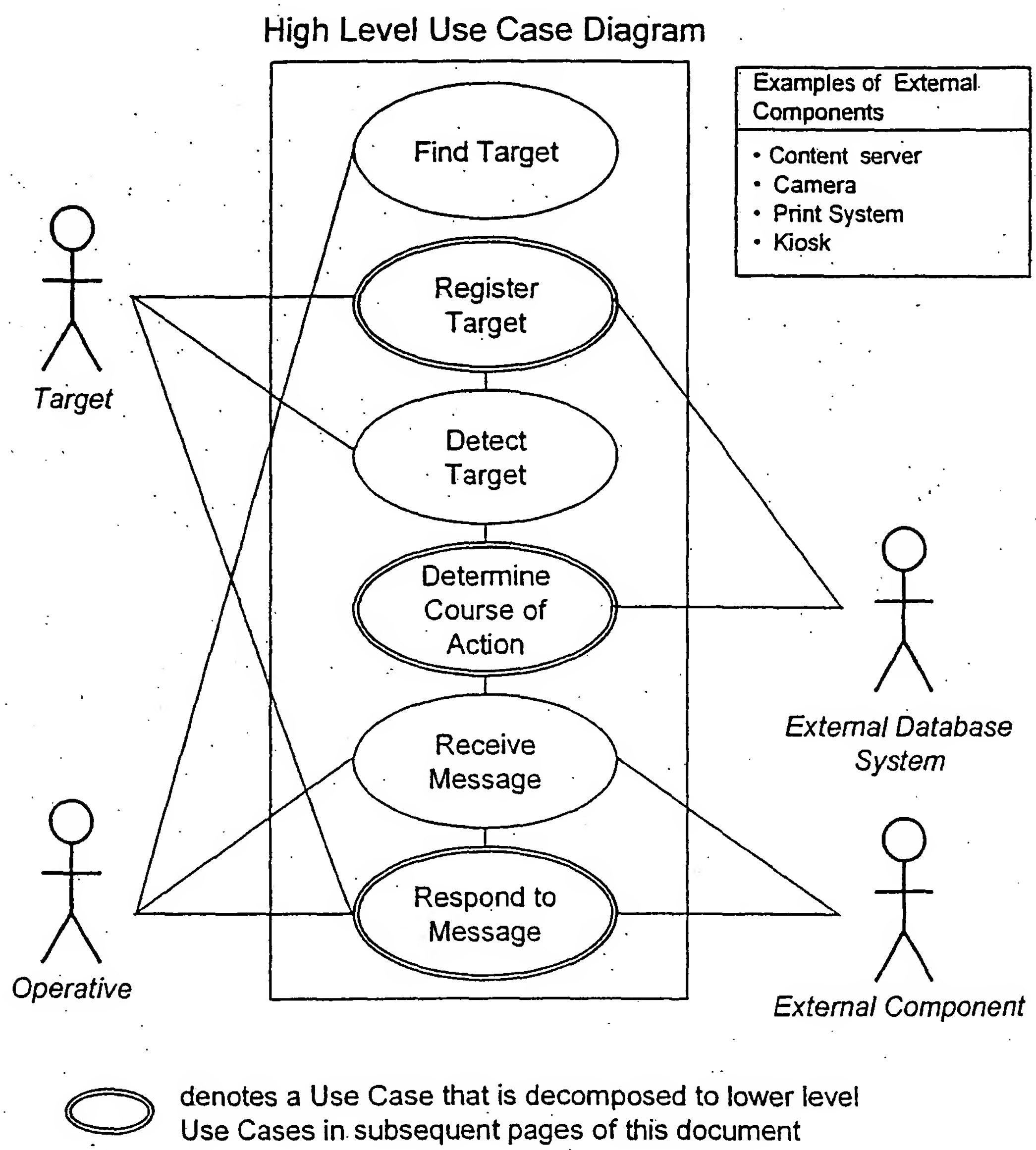
This table defines the actors of the Tabula Rasa service/product offering. An actor is any person or external system that interacts with the system.

Administrator	Person who configures and maintains the system.
External Component	Any external system that interfaces with the system and takes action, such as a kiosk, camera, coupon printer, or gate.
External Database System	Any database that interfaces with the system. Data extracted from an external database system can be imported into the system.
Operative	Person who receives a message when a target is identified.
Target	A person or object that posses an identifier and is registered in the system for the purposes of being monitored.

High Level Use Case Diagram

The Use Case Diagrams describe how the system actors (or users of the system) will interact with the key functions, or use cases of the system. The rectangular box represents the scope of the system, while the ellipses represent the different use cases. The lines drawn from the actors to the use cases indicate interaction points.

Figure 2 below shows an overall view of the system. Some use cases need to be decomposed into several use cases and are indicated as such with the double ellipses.

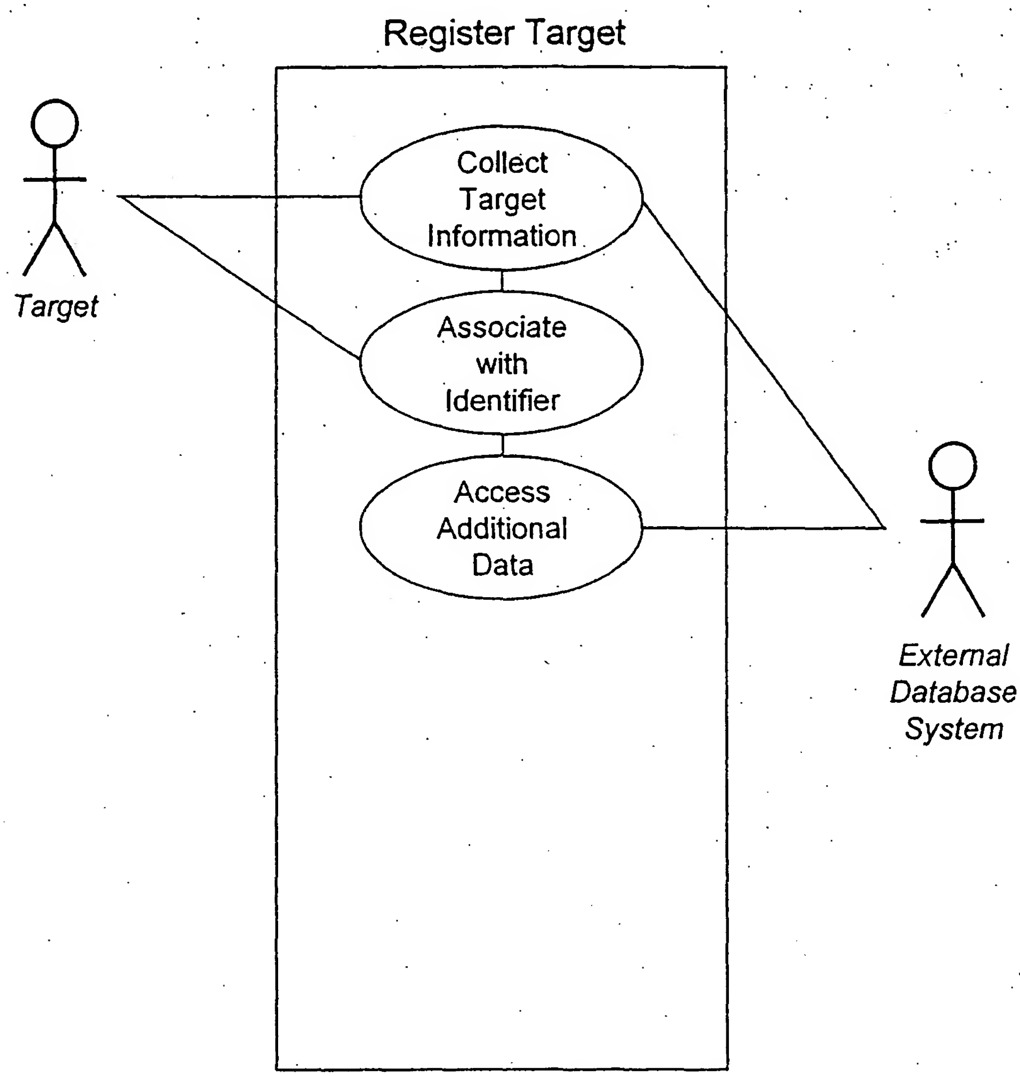


High Level Use Case Diagram

Fig. 2

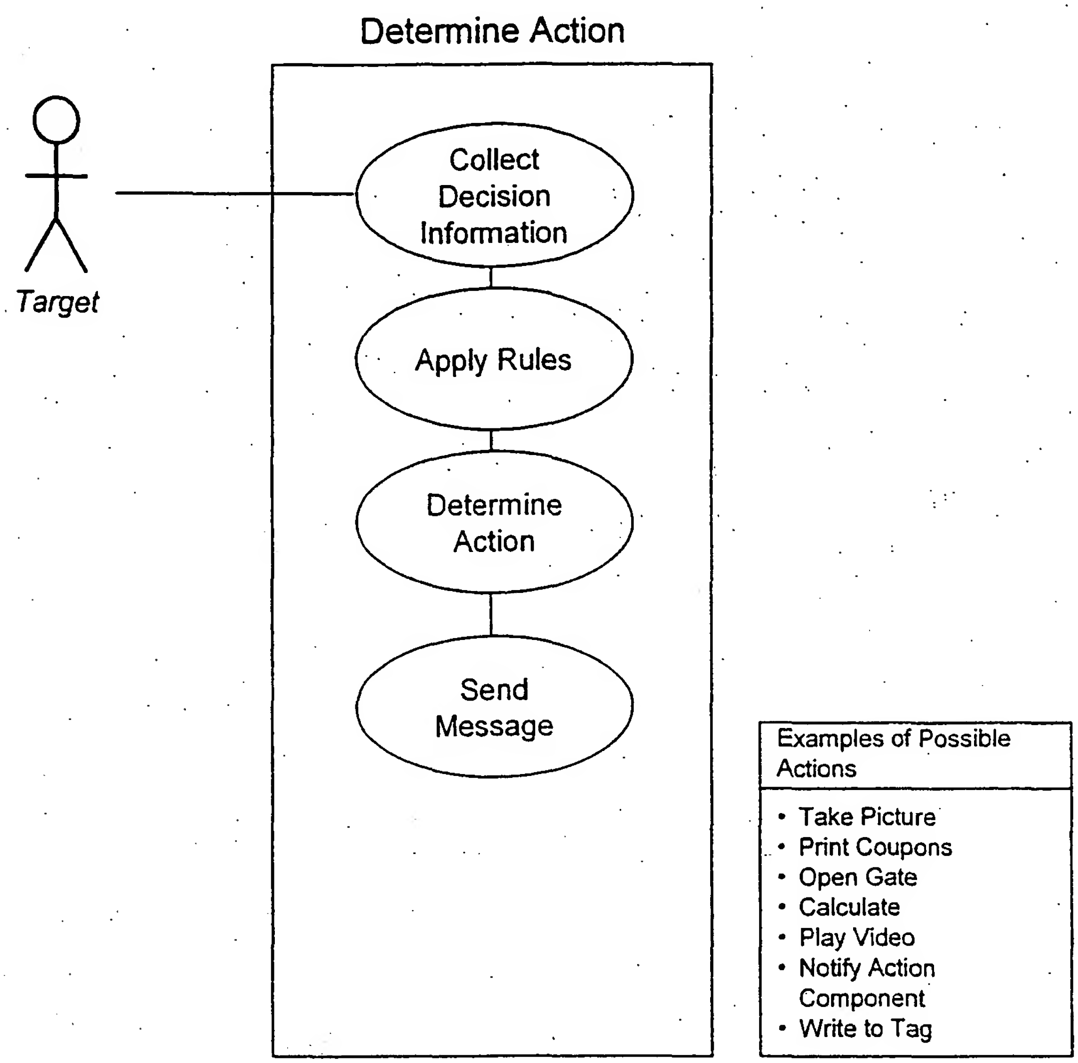
Detailed Use Case Diagrams

The detailed use cases presented below are decomposed from the high-level use cases described in Figure 2 above.



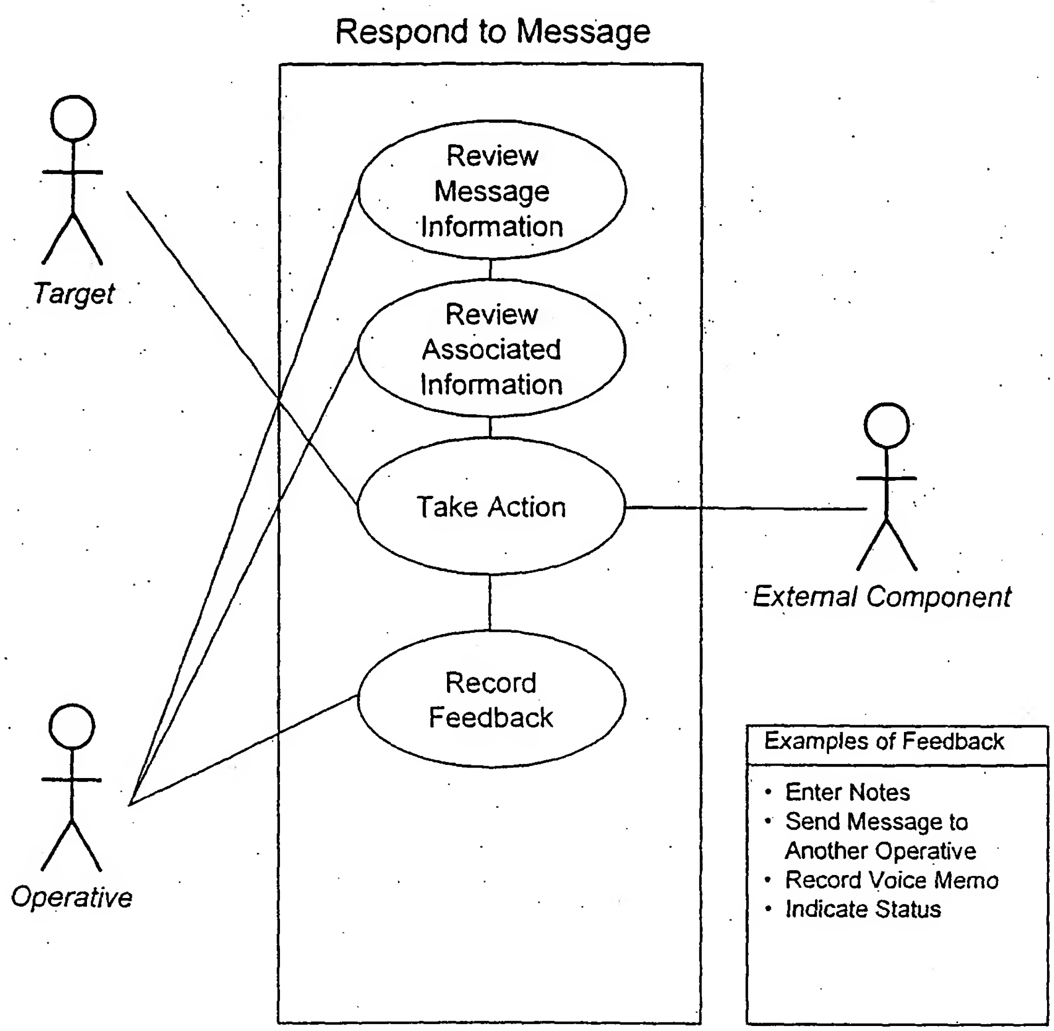
Decomposed Use Case for "Register Target"

Fig. 3



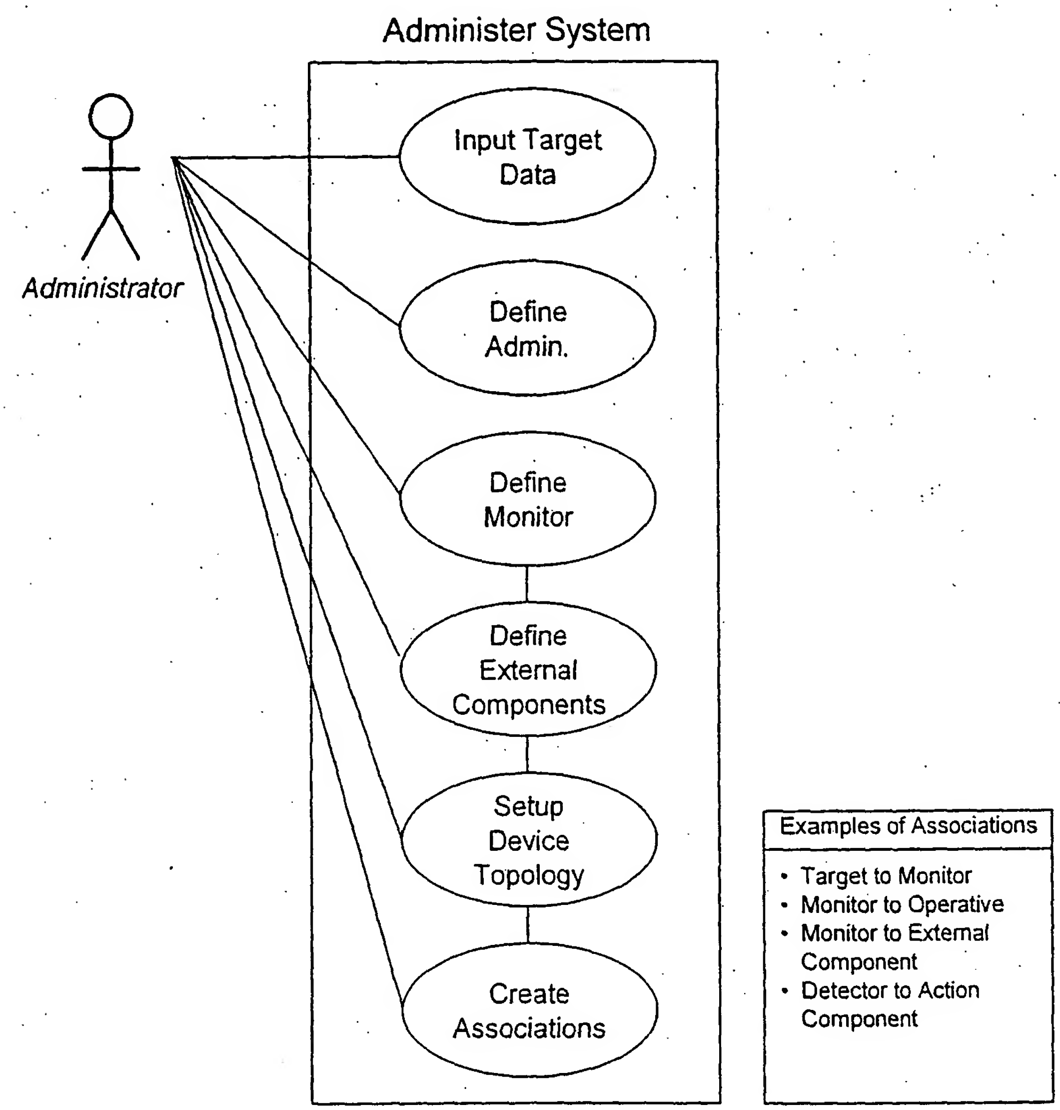
Decomposed Use Case for "Determine Action"

Fig. 4



Decomposed Use Case for "Respond to Message"

Fig. 5



Decomposed Use Case for "Administer System"

Fig. 6

General Component Functions

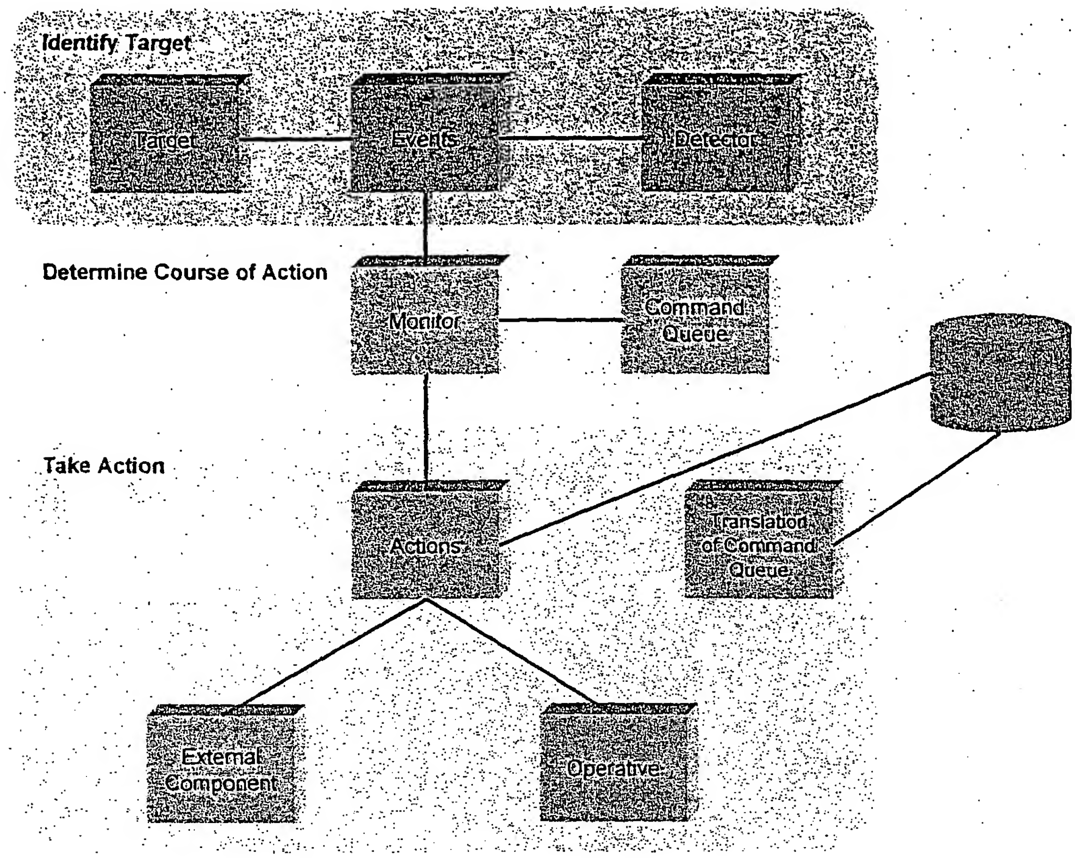
Each high-level use case requires general functions that will be handled by one or more components of the system. The table below lists the general component functions required by each use case, but it does not identify the components themselves. The system architecture will define the components necessary to support the functionality identified here.

High Level Use Case	General Component Functions
Find Target	▪ Query database with criteria ▪ Retrieve query results
Register Target	▪ Receive and format input information ▪ Connect to external database system for additional input information ▪ Associate input information with identifier ▪ Persist target and identifier information
Detect Target	▪ Receive notification of detection ▪ Create and persist event message
Determine Course of Action	▪ Poll for events ▪ Validate ID ▪ Determine action commands based on event messages ▪ Send action command ▪ Persist action command and associated information
Receive Message	▪ Poll for action commands ▪ Acknowledge delivery of action command ▪ Manage action command queue
Respond to Message	▪ Interrogate message ▪ Execute command ▪ Acknowledge command execution ▪ Spawn additional commands ▪ Receive and persist feedback

High Lev l Use Case	General Component Functions
Administer System	▪ Receive and persist administrator data ▪ Import, format, and persist target data ▪ Receive and persist information about monitors ▪ Receive and persist information about devices attached to the system ▪ Receive and persist information about associations between targets, monitors, action components, and detectors

Logical Components

Figure 7 below represents a logical view of the TRCF.



Logical Component Diagram

Fig. 7

The logical component diagram depicts the logical components within the "Identify Target – Determine Course of Action – Take Action" workflow. In the "Identify Target" area, the Target is identified by the Detector and an Event is generated. In the "Determine Course of Action" area, the system Monitor receives the event and determines how to process it and issues a command to the Command Queue. In the "Take Action" area, the Monitor has issued actions and the database is interacted with to determine what the actions mean, based on the command queue. External components are then communicated to, as are operatives and the commands/actions are carried out.

Glossary

Detectors	Device that reads, senses, or scans identifiers.
Device Topology	Logical representation of devices attached to the system, such as a music server, a video streaming device, kiosk, camera, coupon printer, or gate.
Event	Message containing information about a target identification, such as the time or the location of the detector
Identifier	Item used to identify a target, such as an RFID tag or VIP card.
Monitor	Component of the system that recognizes event messages and translates them into business messages.

The purpose of this document is to provide detailed design information about the Tabula Rasa Component Framework (TRCF). This information extends what is provided in the TRCF System Architecture Document and is meant for consumption by engineering personnel who need detailed design knowledge of the individual components in the framework. The format of this document provides for detailed descriptions of each component, and the J2EE principals applied to implement the component.

Extensive use of javadocs via the source code will provide the extra implementation details necessary for a complete understanding of the framework.

Given that many TRCF requirements are fulfilled through the use of other software packages (including open source software), these software packages will be discussed in detail in this document. Examples of such packages include Jakarta's Log4j package for logging debug and information statements to log files. Other software packages used are more than simple support classes. For example, Jakarta's Struts is an entire framework for developing dynamic, data driven web applications. It not only provides a set of support classes, but also dictates its own design methodology. For the sake of clarity, we will duplicate a package's documentation, or refer to it when detailing the design of the overall TRCF.

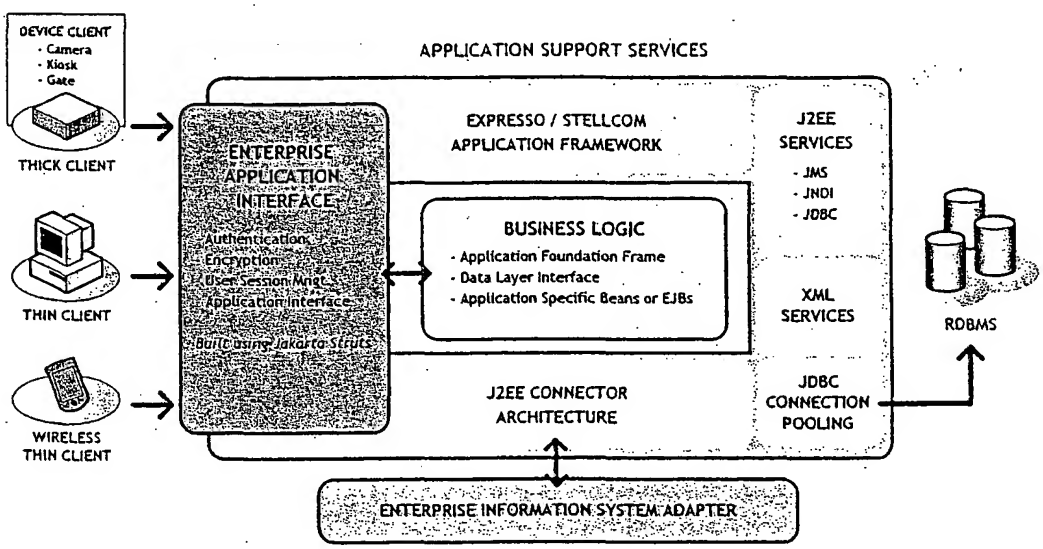
The first thing to clarify is: "What is the TRCF?"

The TRCF is part of a larger system of software. However, it is the core component for processing detection events. These detection events can originate from a variety of sources, yet each is submitted to the TRCF for processing by the event-handling engine built into the TRCF. In its simplest form, the TRCF translates detection events into business processes. These business processes are custom for every deployment of the TRCF, yet the event processing is fundamentally the same for each deployment. When coupled with the specific applications that act on the business processes and consume the detection event information (and related data), you have a complete Tabula Rasa System installation. Initial business ideas have included such ideas as the Marketing Event Manager (MEM) and Casino-based applications. Other possible uses are a system for accessing digital content from a database, for example for selecting music performances for listening.

Design Considerations

The fundamental design principals surrounding the project, are the use of an open, extensible, standards based architecture. As mentioned in the architecture document, this translates into developing as much of the system as possible using Java and J2EE technologies. This section will lead the reader through all of the technologies employed in the design, and the reasons the choices were made.

Java / J2EE



Java / J2EE Component Framework

Fig. 8

Coding Standards

Refer to Stellcom's Java Coding Standard. A copy also resides in the Stellcom KMS system at: http://intraspect.stellcom.com/gm/document-4848322313060353.944663

Development Environment

This document assumes usage of Sun's Forte (NetBeans) for basic IDE functions like editing and debugging. While Forte has a built in version of Ant for project level building, an external installation of Ant will be used to perform scheduled build and execute test suites. More information about the installation and configuration of these and other supporting software is contained in this manual.

Logging

As Brian W. Kernigan and Rob Pike put it in their truly excellent book "The Practice of Programming"

As personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient.

The open source package Log4j is used to log errors in the Tabula Rasa Component Framework. Log4j has three main components: categories, appenders and layouts. These three types of components work together to enable developers to log messages according to message type and priority, and to control at runtime how these messages are formatted and where they are reported.

Log4j makes it easy to name categories by software component. This can be accomplished by statically instantiating a category in each class, with the category name equal to the fully qualified name of the class. This is a useful and straightforward method of defining categories. As the log output bears the name of the generating category, this naming strategy makes it easy to identify the origin of a log message. However, this is only one possible, albeit common, strategy for naming categories. Log4j does not restrict the possible set of categories. The developer is free to name the categories as desired. Nevertheless, naming categories after the class where they are located seems to be the best strategy known so far.

The ability to selectively enable or disable logging requests based on their category is only part of the picture. Log4j allows logging requests to print to multiple destinations. In log4j speak an output destination is called an appender. Refer to the LoggingStandards.txt file in Source.Safe for additional details on logging within the TRCF.

Nested Diagnostic Contexts:

In a multithreaded server environment, distinguishing the log entries by the clients making requests is crucial. With the event-handling engine, the individual clients are the detection devices, which always identify themselves in their requests. However, application user interfaces, such as the administration tools, and the venue specific applications require that logging use the individual's application context so that requests can be distinguished. T

The NDC is managed per thread as a stack of contextual information. Note that all methods of the org.apache.log4j.NDC class are static. Assuming that NDC printing is turned on, every time a log request is made, the appropriate log4j component will include the entire NDC stack for the current thread in the log output. This is done without the intervention of the user, who is responsible only for placing the correct information in the NDC by using the push and pop methods at a few well-defined points in the code.

Error Handling

The appropriate action depends on whether the code that experiences an error can resolve the problem. The next question is whether the calling code may be able to resolve the problem (perhaps by retrying). If it can, then you can simply throw the appropriate exception to your caller. If you can't find an appropriate exception in the Java class libraries, feel free to make a new one or use a StellcomException. If the calling code can't resolve the problem, the next question is whether the system that it's part of can continue in any useful fashion. For a standalone app, the answer is often no - for example, an app that is supposed to listen on a particular TCP/IP socket may just as well shut down if it can't open that socket (with an informative message to the user of course). When the code is part of a larger system, particularly when it's in an app server that may also be supporting other applications we don't even know about, shutting everything down is rarely an acceptable option.

This is where the StellcomException comes in, because it can be used to guarantee that the error will eventually be logged & handled somehow. Even if a subsystem is incapable of functioning the other subsystems can continue on. Also, it leaves the door open for some other piece of the system to attempt recovery. For example, if the app in an app server can't find something it needs in the JNDI tree, it might send an e-mail to the system administrator asking him to add an appropriate entry & then go the admin screen for this app & click a Retry button.

Security

Security is a process ensuring that both access to the system, and authorization to perform certain tasks within the system are controlled. The security requirements of your TRCF system will depend greatly on what type of environment the system is deployed. If the system is exposed directly to the Internet, or is only deployed in an isolated environment may influence the number of steps taken to ensure security.

There are generally 3 access points provided by the TRCF – Physical access, Event Source Manager, and User/Administrator Web Applications

Physical access

Users who can login to the TRCF server(s) and perform operating system functions must be controlled. This involves managing operating system specific usernames and passwords. General security principals for the given platform must be applied and maintained.

Event Source Manager

New events are entered into the system through a servlet. Only those detection devices that are authorized may enter new events in the system. The device must supply the appropriate credentials when supplying a new event, and the Event Source Manager must verify those credentials against a list of known (registered) devices. In the first release of the product, this will consist of verifying that the device (id) is "registered" in the database.

User Interface Programs

There are 2 general user applications used in the TRCF. A web application is used for administering the TRCF, and business-specific web applications are used to allow businesses to interact with the system. Both applications use the Struts framework, which maintains credentials for all users that access the system. Any application must provide credentials via the exposed Struts-derived application interface. This takes the form of a username and password prompts, combined with session management.

General Security Principals

Security Is a Process

What this means is that any application server should be routinely audited, and there will always be new security enhancements available. Routine updates, audits and patches will always be needed. Engineering Support of new configurations, and Customer Support of existing installations must work together to ensure timely updates for customers.

Run your application under the most recent version of the JVM as possible - There are always bug fixes that are included in every new release, and some old bugs may be able to be exploited by a hacker.

Install strong encryption. Strong encryption helps prevent attackers from stealing the actual passwords that you may use to log into your system. This may involve simple use of HTTPS to prevent the sniffing of usernames and passwords from the network.

Run your struts setup under a java security policy - As of JDK version 1.2, Java has some extremely fine-grained security possibilities. Things to do with the policy include but are not limited to:

- Limiting the classes that can access the file system to only those necessary. This will mainly be limited to the configuration classes and that should be it.

- Limiting the classes that can make network connection. You also have the ability to determine what IP address you will allow connection from. This is very good if you are running your application server as dependant on another web server (See the Apache Jakarta project for what I mean by this). By and large, you'll only want connections coming in directly from the web server. You'll also want to make sure that only the JDBC client classes, and the classes responsible for connections to the web server are the only classes allowed to make outbound connections. This way a hacker cannot add a small class that sends passwords directly to his own site.

See your JDK documentation for more information on creating a policy file.

Secure the file system. Make sure that the only users that have access to the Application server directories are root for full control, and the account that the Application Server is going to run under for read only and execute control. If you have some directories where the Application Server is going to write files, open that directory, but remember, whenever a file can be written to your hard drive is a point of entry that a hacker may come in through. So keep the file system permissions limited. Also make sure that you have completely restricted access to the Struts configuration directory.

Secure the database JDBC drivers. Depending on what machine the database is located on, this may be different than the "Run under a security policy" step. Make sure that the db drivers only accept connections from certain sources, and block out all other connection attempts.

Employ Firewalls between the Internet and your database. Protect your database with every available means.

Run the database user with restricted privileges so that there are only permissions granted to the app that coincides with what is absolutely necessary to run.

Remove all unnecessary services: Some servlets and services may not be used in a production environment. Remove every service on your application server that is not required for the web site to function properly.

Remove all default passwords. The Openhack Project got cracked simply because somebody forgot to change a default password that comes with Oracle. Make sure you change all passwords for default user accounts. Or better yet, create an Administrative account under a different name, and then delete the default user accounts all together. That way a hacker has to start from scratch.

Run security-monitoring applications: Most determined hackers would eventually get in. It's just a matter of time. It's best to have a series of intrusion detection systems in place. This way you can get notified if suspicious activity is going on. Or if you have the budget for it, check out the monitoring service that Counterpane Internet Security offers.

Run External Audits: It's next to impossible to see your own mistakes. Have a security firm (or two) do at least a basic audit of your web server system. This will at least catch some of the more common mistakes you might make in your installation.

Licensing

The TRCF is designed using a variety of open source software packages. These packages have licensing constraints that must be honored when delivering the TRCF software. Please refer to the Licensing Appendix at the end of this document for the current copies of these software license agreements.

Other software packages, specifically those used in the high-end configuration require licensing to run the products. Please refer to the specific vendors licensing documentation for further details.

Clustering & Scaling

JBOSS

No clustering will be available in the JBOSS configuration. Only vertical scaling is available – via the purchase of higher-end hardware. Note that the JBOSS we site indicates how to achieve "clustering" with their current release, but it is a manual workaround (at best).

Weblogic

At the time of this writing, Weblogic 6.1 supported clustering in JMS, yet did not support fail over. There is still a single point of failure in that a single server contains the JMS queue and topic definitions. If that server is lost, the system will cease to function. Furthermore, the loss of other servers in the cluster will result in a loss of some messages. It is hoped that the next release of Weblogic (7.0) will include better support and implementation of JMS clustering.

Reliability / Fault Tolerance

Device Manager restart

DM clients are configured to persist their configuration settings locally, and startup automatically when their machine reboots. Any detection events, which occur prior to full startup, will be lost. Once startup has completed, events will resume propagation to the server.

Application Server restart / Software restart

The application server is configured to launch the application automatically on a system restart. This ensures system recovery during any intermittent power loss, or simply when the system is brought up. DM clients cannot communicate with the server during the startup. DM clients will require additional design to cache events during times the application server is not available. JMS persistence will preserve any messages that have not yet been delivered.

Database Down, Deadlocks, and other message delivery failures

For Action Components that require interaction with external systems (such as a database), that may fail, use of transactions will ensure that any error in message consumption causes a rollback to occur so that messages can be redelivered. Failure to deliver messages in a configurable number of attempts will result in the message being delivered to a Dead Letter Queue (DLQ) for the message to be manually addressed once system failures have been repaired.

JBOSS

Given the lack of clustering in the low-end JBOSS configuration, the reliability and fault tolerance is less than the high-end WebLogic configuration. The specific implementation of message redelivery on JBOSS works as follows (from the source code):

Places redeliveded messages on a Dead Letter Queue. The Dead Letter Queue handler is used to not set JBoss in an endles loop when a message is resent on and on due to transaction rollback for message receipt. It sends message to a dead letter queue (configurable, defaults to queue/DLQ) when the message has been resent a configurable amount of times, defaults to 10. The handler is configured through the element MDBConfig in container-invoker-conf. The JMS property JBOSS_ORIG_DESTINATION in the resent message is set to the name of the original destination (Destionation.toString()). The JMS property JBOSS_ORIG_MESSAGEID in the resent message is set to the id of the original message.

What this means, is that for simple, recoverable errors, retrying the message send allows the system to complete delivery without error. For problems that are more lengthy (database down , etc.), requiring administrative involvement, the message is attempted N times, then moved to a

storage place to be manually dealt with once the system error has been resolved. In this case, the administrator will use appropriate tools to resend the queued messages.

WebLogic

The application server is configured to have the same functionality as JBOSS, for using DLQ.

JMS

Java Message Service providers are used by the TRCF for the delivery of Event and Command messages. The Java Message Service enables distributed communication that is:

- Loosely Coupled
- Asynchronous
- Reliable

Features

The JMS API in the J2EE 1.3 platform has the following features:

- Application clients, Enterprise JavaBeans™ (EJB™) components, and web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously.
- A new kind of enterprise bean, the message-driven bean (MDB), enables the asynchronous consumption of messages. A JMS provider may optionally implement concurrent processing of messages by message-driven beans. TRCF components will receive messages asynchronously using MDB.
- Message sends and receives can participate in distributed transactions. Transactions will be used by the Monitor & Action Commander to consume and submit messages in response to a given detection event. Action Components will employ transactions given the specific business requirements, to ensure that message delivery is handled in a fault tolerant manner.

JMS Application

A JMS application is composed of the following parts:

- A JMS provider is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform at release 1.3 and above includes a JMS provider. Both JBoss and WebLogic provide a JMS implementation
- JMS clients are the programs or components written in the Java programming language that produce and consume messages. In the Tabula Rasa Component Framework, the Event Source Manager and Action Commander produces messages and the Monitor and Action Components consume messages

- Messages are the objects that communicate information between JMS clients. Messages in the TRCF will be constructed from serialized objects
- Administered objects are preconfigured JMS objects created by an administrator for the use of clients. There are two kinds of administered objects, destinations and connection factories, which are described in "Administered Objects". These objects will be primarily created through the use of deployment descriptors. Deploying new objects that specify new topics may require restarting the application.
- Point to Point and Publish / Subscribe domains. The TRCF employs only the Publish / Subscribe model when moving messages through the system.

Publish Subscribe

General Features of Publish / Subscribe

- Each message may have multiple consumers - message can be processed by zero, one, or many consumers.
- There is a timing dependency between publishers and subscribers, because a client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

Message Consumption Methods

Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives, or it can time out if a message does not arrive within a specified time limit.

Asynchronously: A client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage method, which acts on the contents of the message.

The TRCF will employ Message Driven Beans, which by design use the asynchronous method listed above.

JMS application building blocks

Administered objects (connection factories and destinations)

Connections

Sessions

Message producers

Message consumers

Messages

A **connection factory** is an object a client uses to create a connection with a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.

Context ctx = new InitialContext();

TopicConnectionFactory topicConnectionFactory = (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");

Given the differences between topic connection factory names between JBOSS and WebLogic, a dynamic method of selecting the name is employed.

A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes.

Topic myTopic = (Topic) ctx.lookup("topic/EventTopic");

These are the topics where the Action Commander will publish messages

A **connection** encapsulates a virtual connection with a JMS provider. It could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();

When an application completes, you need to close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

topicConnection.close();

A **session** is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages.

Sessions serialize the execution of message listeners. A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

TopicSession topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully. Note that for message consumers such as TRCF Action Components that perform tasks that may fail (such as inserting a row in a database), use of JMS transactions is mandatory to ensure that messages are not lost.

A **message producer** is an object created by a session that is used for sending messages to a destination.

TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);

Once you have created a message producer, you can use it to send messages.

topicPublisher.publish(message);

A **message consum r** is an object created by a session that is used for receiving messages sent to a destination. A messaoe consumer allows a JMS client to register interest in a

destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);

You use the TopicSession.createDurableSubscriber method to create a durable topic subscriber.

Once you have created a message consumer, it becomes active, and you can use it to receive messages. Message delivery does not begin until you start the connection you created by calling the start method. To consume a message asynchronously, you use a message listener. This is the method employed automatically by a message driven bean.

A **message listener** is an object that acts as an asynchronous event handler for messages. It implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.
TopicListener topicListener = new TopicListener();

topicSubscriber.setMessageListener(topicListener);

When using message-driven beans, the listener is automatically assigned.

Filtering messages

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than the application. A message selector is a String that contains an expression. The syntax ~~of the~~ expression is based on a subset of the SQL92 conditional expression syntax. The createReceiver, createSubscriber, or createDurableSubscriber methods each have a form that ~~allows you~~ to specify a message selector as an argument when you create a message consumer. The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages based on the content of the message body. Message-driven beans specify the filter string in the deployment descriptor. The Action Commander default behavior is to map messages from device types to Action Components (AC). In the initial release, a single command topic is used to send messages to ACs, using filters to specify which AC is to receive the message (given that they all "watch" the same JMS topic). The previous statement is the desired design, however, due to memory leak problems in the JBOSS application server, each Action Component will be given their own topic until the bug can be fixed.

Messages

A JMS message has three parts:

Header

A JMS message header contains a number of predefined fields, which contain values that both clients and providers use to identify and route messages. For example, every message has a unique identifier, represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or topic to which the message is sent. Other fields include a timestamp and a priority level. Each header field has associated setter and getter

methods, which are documented in the description of the Message interface. A client sets some header fields, but many are set automatically by the send or publish method, which overrides any client-set values.

Properties (optional)

You can create and set properties for messages if you need values in addition to those provided by the header fields. The Action Commander uses a "Command" property to specify which Action Component is to receive the message. In the case where no filtering is required, the filter string will be blank

Body (optional)

The JMS API defines five different message body formats, also called message types, which allow you to send and receive data in many different forms. The TRCF uses the ObjectMessage type containing a DetectionEvent object.

JMS Basic Message Reliability

Many JMS applications cannot tolerate dropped or duplicate messages and require that every message be received once and only once. The most reliable way to produce a message is to send a PERSISTENT message within a transaction. JMS messages are PERSISTENT by default. A transaction is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail. The most reliable way to consume a message is to do so within a transaction, either from a non-temporary queue (in the PTP messaging domain) or from a durable subscription (in the pub/sub messaging domain). For other applications, a lower level of reliability can reduce overhead and improve performance. You can send messages with varying priority levels, and you can set them to expire after a certain length of time

The basic mechanisms for achieving or affecting reliable message delivery are as follows:

Controlling message acknowledgment:

Because this can be controlled dynamically, the integrator must be allowed to administratively set what level of acknowledgement an Action Component employs.

Session.AUTO_ACKNOWLEDGE: The session automatically acknowledges a client's receipt of a message either when the client has successfully returned from a call to receive or when the MessageListener it has called to process the message returns successfully. This is the form most often used by the TRCF.

Session.CLIENT_ACKNOWLEDGE: A client acknowledges a message by calling the message's acknowledge method. In this mode, acknowledgment takes place on the session level: acknowledging a consumed message automatically acknowledges the receipt of all messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

Session.DUPS_OK_ACKNOWLEDGE: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should only be used by consumers that can tolerate duplicate

messages. (If the JMS provider redelivers a message, it must set the value of the JMSRedelivered message header to true.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

A business case must be very tolerant of duplicate messages before usage in the TRCF

The provider also retains unacknowledged messages for a terminated TopicSession with a durable TopicSubscriber. Unacknowledged messages for a nondurable TopicSubscriber are dropped when the session is closed.

Specifying message persistence:

The JMS API supports two delivery modes for messages, which specify whether messages are lost if the JMS provider fails.

The PERSISTENT delivery mode, which is the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.

The NON_PERSISTENT delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

The TRCF uses the default of PERSISTENT for all message delivery. This is specified in the deployment descriptor for the application server.

Setting message priority levels:

You can set different priority levels for messages, which can affect the order in which the messages are delivered. It is unclear as to the needs of the TRCF if priorities are required. Do ~~we at~~ least need the delivery to be chronological?

Allowing messages to expire:

~~You can specify an expiration~~ time for ~~messages~~, so that they will not be delivered if they ~~are~~ obsolete. If the TRCF requires this to be dynamically set, administration interfaces are required to set this, and code written to dynamically set the time to live. The first version of the product will not support this feature.

Creating temporary destinations

The TRCF will not create temporary destinations. The use of deployment descriptors may result in the application server creating what it thinks is a temporary destination

JMS Advanced Reliability

The more advanced mechanisms for achieving reliable message delivery are the following:

Use message persistence

To make sure that a pub/sub application receives all published messages, use both PERSISTENT delivery mode for the publishers and durable subscriptions for the subscribers

Creating durable subscriptions:

You can create durable topic subscriptions, which receive messages published while the subscriber is not active. Durable subscriptions offer the reliability of queues to the publish/subscribe message domain. Note that since all subscribers (Action Component MDBs) are always active, durability is not used by default.

Using local transactions:

You can use local transactions, which allow you to group a series of sends and receive into an atomic unit of work. Transactions are rolled back if they fail at any time. Application server-specific methods of dealing with rollbacks, and resends are documented in the Reliability / Fault Tolerance section of this manual.

JMS in J2EE

The Platform Specification recommends that you use java:comp/env/jms as the environment subcontext for Java Naming and Directory Interface (JNDI) lookups of connection factories and destinations.

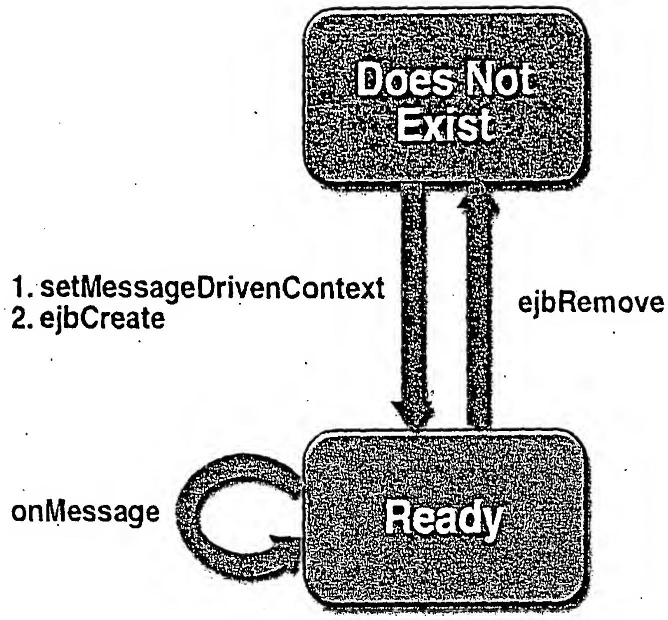
Instead of looking up a JMS API connection factory or destination each time it is used in a method, it is recommended that you look up these instances once in the enterprise bean's ejbCreate method and cache them for the lifetime of the enterprise bean.

If you wish to maintain a JMS API resource only for the life span of a business method, it is a good idea to close the resource in a finally block within the method. If you would like to maintain ~~a~~ JMS API resource for the life span of an enterprise bean instance, it is ~~a good idea to use the~~ component's ejbCreate method to create the resource, and to use the component's ejbRemove ~~method~~ to close the resource. If you use a stateful session bean or an ~~entity bean and you wish~~ to maintain the JMS API resource in a cached state, you must close the resource in the ejbPassivate method and set its value to null, and you must create it again in the ejbActivate method.

Instead of using local transactions, you use the deploytool to specify container-managed transactions for bean methods that perform send and receive, allowing the EJB container to handle transaction demarcation. (You can use bean-managed transactions and the javax.transaction.UserTransaction interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions. Usually container-managed transactions produce the most efficient and correct behavior.)

Message Driven Beans (MDB)

The following illustrates the stages in the life cycle of a message-driven bean



The EJB container automatically performs several setup tasks that a standalone client has to do:

- Creating a message consumer (a QueueReceiver or TopicSubscriber) ~~to receive the~~ messages. You associate the message-driven bean with a destination and connection factory at deployment time. If you want to specify a durable subscription or use a message selector, you do this at deployment time also.
- Registering the message listener (you must not call setMessageListener)
- Specifying a message acknowledgment mode
- Your bean class must implement the javax.ejb.MessageDrivenBean and javax.jms.MessageListener interfaces.
- Your bean class must implement the ejbCreate method in addition to the onMessage method. The method has the following signature:

  public void ejbCreate() {}
- If your message-driven bean produces messages or does synchronous receives from some other destination, you use its ejbCreate method to look up JMS API connection factories and destinations and create the JMS API connection.
- Your bean class must implement an ejbRemove method. The method has the following signature:

  public void ejbRemove() {}
- If you used the message-driven bean's ejbCreate method to create the JMS API connection, you ordinarily use the ejbRemove method to close the connection.

- Your bean class must implement the setMessageDrivenContext method. A MessageDrivenContext object provides some additional methods that you can use for transaction management. The method has the following signature:

  public void setMessageDrivenContext(MessageDrivenContext mdc) {}

There are 2 places in the TRCF where JMS topics are employed to handle the movement of events and commands through the framework. The first is an Event Topic and the second ~~is a~~ Command Topic. The clients who receive messages from these topics are the Monitor and the Action Component (respectively). When choosing to use a MDB, we consider the following related concepts:

Connection Factories – These are application-server-specific and while users can create their own, it helps to understand the underlying features of a connection factory. For example, in Jboss a java:/ConnectionFactory is a factory type of INVM and is the fastest factory while ConnectionFactory is a factory type of OIL which has other characteristics

Jboss Connection Factories:

Destination type	**JNDI name**	**Factory type**
Topic/Queue	`RMIConnectionFactory`	RMI
Topic/Queue	`RMIXAConnectionFactory`	RMI. Supports XA transaction
Topic/Queue	`java:/ConnectionFactory`	INVM
Topic/Queue	`java:/XAConnectionFactory`	INVM. Supports XA transaction
Topic/Queue	`ConnectionFactory`	OIL
Topic/Queue	`XAConnectionFactory`	OIL. Supports XA transaction
Topic/Queue	`UILConnectionFactory`	UIL
Topic/Queue	`UILXAConnectionFactory`	UIL. Supports XA transaction

Message Durability – Use of durability must be factored by the business process performed. Some processes may require durable subscriptions, others may not.

Transactions – Transactions will be required by the message producers and consumers in the TRCF to ensure messages are not lost if delivery or consumption fails.

Message Persistence – The TRCF will persist messages by default. The persistence mechanism is application-server-specific and each has varying reliability and ~~performance~~ issues. For example, in Jboss, the File Persistence Manager is very stable but not very fast. The Rolling logged is very fast but is newer, and less proven. Regardless if messages are persisted in a file system file, or a database, there are pros and cons to the implementation.

Jboss Persistence Managers

PM Name	**Advantages**	**Disadvantages**
`File`	Very Stable	Not fast
`Rolling logged`	Very Fast (default)	Very new, somewhat unproven
`JDBC`	Should provide better stability/scalability	Has JDBC overhead

Logged	Fast	Log files grow without bound, memory management problems during recovery

Topics – The number of topics used in the system will vary. By default, there is only one Event Topic and one Command Topic. Message-driven beans use message filtering to pull their messages from the topic. The message must container a header value equal to the MDB class name.

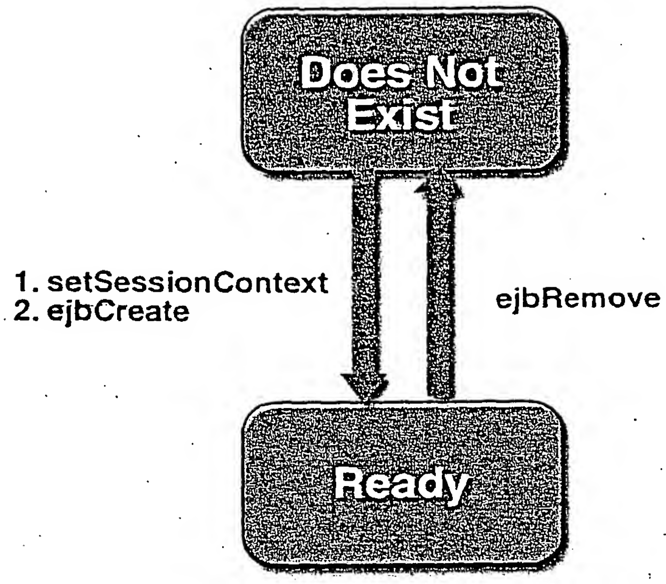
Message Formats – Serialized objects will be used to send messages through the system

Unit Tests – Test for JMS use simple client classes that inject messages into the specific topic for consumption by the registered MDB. The test message will instruct the listener to log information that will prove the system is functioning at various levels.

Administration – The user interface required to administer the JMS-related items. In the TRCF, the only administration supported involves associating device types with action components.

Stateless Session Beans

Besides Message Driven Beans, the other main type of EJB employed in the TRCF is the Stateless Session Bean. Because a stateless session bean is never passivated, its life cycle has just two stages: nonexistent and ready for the invocation of business methods. The following illustrates the stages of a stateless session bean



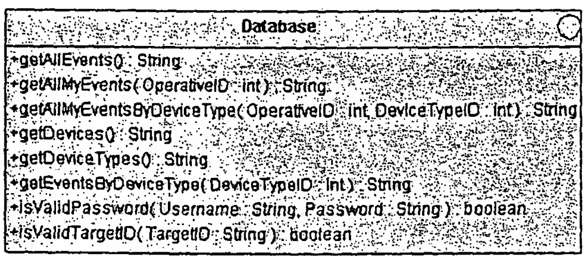
Component Design

Common Components

Please refer to the Javadocs for detailed information on these components, or for those not listed specifically.

Database Interface

There will be two general interfaces into the internal TRCF database system. One will be via Action Components that will perform specific database inserts, updates, and selects based on the business requirements for a venue-specific implementation. The other interface will consist of those operations considered standard for the TRCF. This includes many of the inserts, updates, deletes, selects required for operations such as system administration, general venue administration, and generic host application information retrieval. The implementation is via stateless session bean using JDBC connections. The required operations will be exposed as specific methods in the stateless session bean.



Sending JMS messages

There are two general message producers in the TRCF, including the Event Source Manager and the Action Commander. In addition, some test classes produce messages to inject in the system during testing. This common need is satisfied with a general SendDetectionEvent class.

Application Properties

For those classes that persist properties in a file, the ApplicationProperties class allows for reading and writing properties.

DetectionEvent ObjectMessage

All messages processed by the event process engine travel through the various topics and components as a JMS ObjectMessage. There are a fixed number of member variables that are set once when the message is constructed. The remaining message producers can only make changes to the message by applying custom header properties.

Standard Components

Tabula Rasa Component Relationships

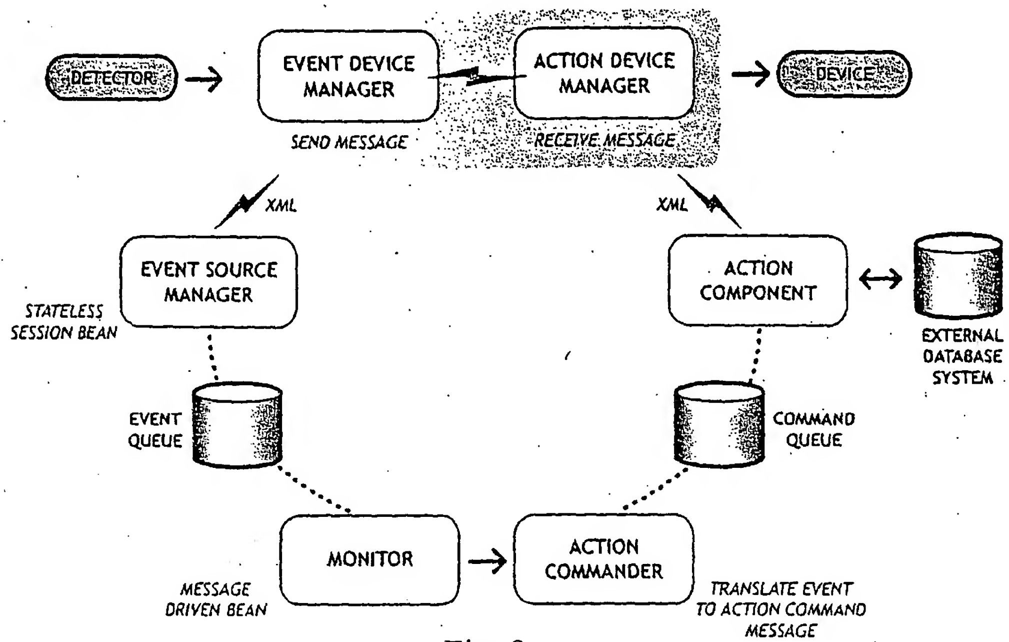


Fig. 9

Device Manager

~~The Device Manager is the subsystem responsible for talking~~ to physical ~~devices connected to~~ the system. System devices are split into two basic categories: the devices responsible for capturing system events and the devices that respond to system events by performing some ~~action~~. The classes that communicate with ~~these devices will be customized subclasses of the~~ DetectionDevice and Device classes respectively. For convenience, DectionDevice is a subclass of Device allowing a single class to manage a device that both detects and responds to events.

In the normal situation, a DetectionDevice class will receive an event from the device it is managing and hands this event to the EventProcessor, then immediately goes back to receiving new events. The EventProcessor queues these events and in a separate thread ~~relays them to~~ any registered observers. The EventProcessor implements the RemoteObservable interface, and therefore can be observed by any class implementing the RemoterObeserver interface. One such class is the SystemProxy. The SystemProxy builds an XML message based ~~on the~~ event and posts it via HTTP to the server. From there the event enters the Tabula Rasa framework. In this case, the responding devices will receive notification of the event when an ActionComponent calls the requestInformation() method.

The SystemProxy will be a part of all systems, allowing the events to enter the Tabula Rasa framework. However situations may arise in which responding devices must be informed of the events immediately, without introducing the latency inherent to having them pass through ~~the~~ framework first. For this reason, Devices can also register to observe DetectionDevices ~~directly.~~

Since this observation can occur over Java RMI, the Devices do not have to be in the same process or even on the same machine as the DetectionDevices, provided that they can communicate over a network. The details of this RMI communication are handled by the DetectionDevice and Device classes. Concrete device manager subclasses will not have to be concerned with them.

The basic class structure is as follows:

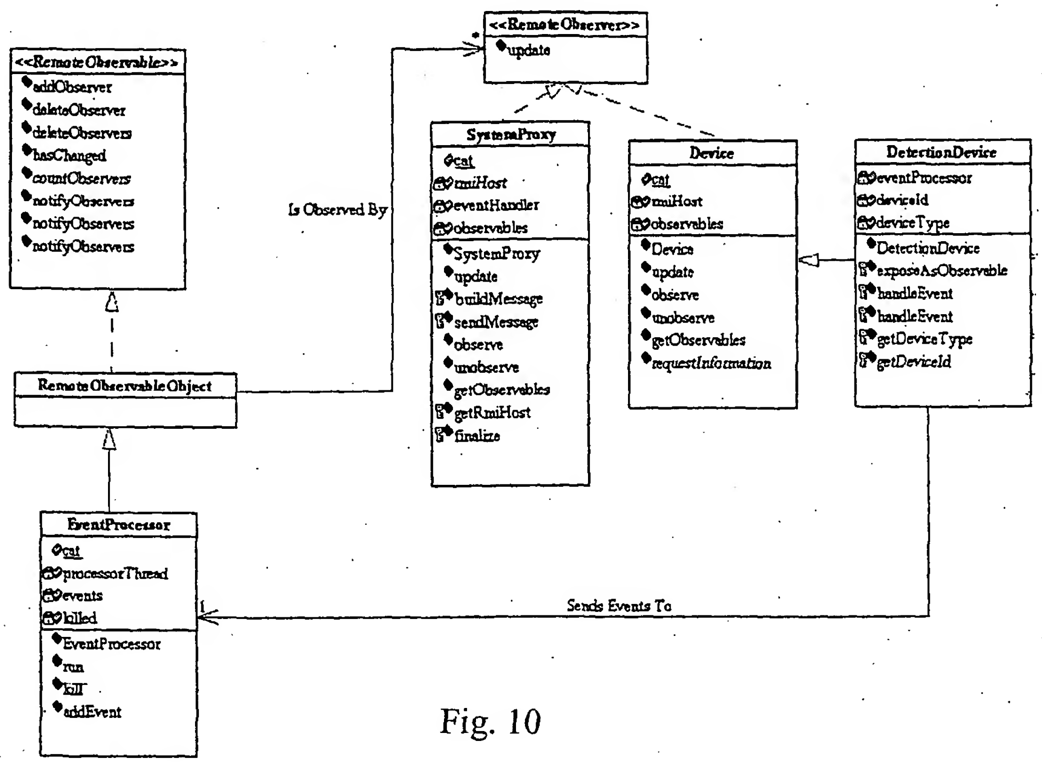


Fig. 10

Since the communication from the Device Manager subsystem to the Tabula Rasa framework is through an XML string posted via HTTP, the Device Manager could in theory be implemented in any language. The format of the XML is as follows:

```
<detectionevent>
    <targetid>
        85ABC00000000010
    </targetid>
    <devicetype>
        1
    </devicetype>
    <deviceid>
        1
    </deviceid>
    <timestamp>
        1007500148365
    </timestamp>
</detectionevent>
```

Refer to the devicemanager.properties file for a complete list of the properties used by the device manager. The following properties are persisted for use by the device manager process when it starts

hostname – the IP Address or host name of the machine running the RMI Registry (defaults to localhost)

port – the port the RMI Registry runs on (defaults to 1099, which is probably OK 99% of the time)

eventhandler – the URL of the servlet that receives events and hands them to event source manager. This could change if we start using SOAP instead of posting an XML string.

deviceid – the id to use in messages transmitted to the server.

devicetype – the device type id to use in messages transmitted to the server.

Omron Reader (hardware specific):

comport -- the com port string, like "COM1" or "COM2"

tagdelay -- the delay in ms before allowing the same tag to be scanned again, like "2000"

Event Source Manager

The Event Source Manager (ESM) receives messages from the various ~~Device Manager~~ processes deployed at a given venue. It is implemented as a servlet called EventPipe which uses a stateless session bean EventManagerEJB to parse the xml-formatted message and ~~publish it to the~~ default EventTopic. This design allows for input of a large ~~number of input messages,~~ and separates the message receiving from message ~~processing.~~ ~~The ESM returns~~ standard HTML to the client device manager. The design of the ESM must be kept lean to allow processing messages at a very fast rate. The ESM relies on a JMS topic (EventTopic) to persist the incoming messages to that it can return to receiving more messages.

Monitor & Action Commander

Incoming Messages

These messages come from the Event Source Manager every time an event is detected. The Event Source Manager is responsible for processing messages from devices and storing ~~the~~ extracted data in an instance of the DetectionEvent class. (Note that the DetectionEvent class shown in the TRCF System Architecture 1.0 doc does not have a device ID field, and we need to add one.)

The incoming messages will be JMS ObjectMessages, containing the DetectionEvents.

Monitor

A straightforward Message Driven Bean (EJB) that listens for incoming messages. When one arrives the Monitor extracts the DetectionEvent object & passes it to the ActionCommander. There may be multiple instances of the EJB within the app server.

Action Commander

This component is invoked by the Monitor. Essentially it receives a message about an event and in response sends out one or more messages to the Action Components. The API is

ActionCommander.processEventMessage(DetectionEvent oDetectionEvent)

The outgoing messages are determined by reading the Configuration Repository (discussed next).

Configuration Repository

The Action Commander reads from here to determine its behavior. The configuration is a series of mappings that connect one source to one to many destinations. A source is distinguished by a device type and an optional device ID. A destination consists of a JMS topic name, a string that goes in the JMS header, and a string that goes in the message body. Reading this data from a repository should allow us to accommodate almost any future need.

Device ID is optional so that all devices of a particular type can cause the same outgoing message. If you want to have two specific devices cause the same outgoing message, you put two entries in the configuration. If you want one device to cause more messages than other devices of that type you can have one entry with no device ID & another with a specific device ID that causes extra messages. There is no simple way to make one device cause fewer messages than other devices of that type.

The configuration repository will be an XML file. If possible the Action Commander will include a function that forces the reloading of the configuration file at run time.

Here is the DTD for the configuration repository (if time permits I want to try using a schema instead):

```
<!ELEMENT DeviceID>
<!ELEMENT DeviceType>
<!ELEMENT Source (DeviceType, DeviceID?) >
<!ELEMENT JMSTopicName>
<!ELEMENT HeaderString>
<!ELEMENT MessageString>
<!ELEMENT Destination (JMSTopicName, HeaderString, MessageString) >
<!ELEMENT CommandMapping (Source, Destination+) >
<!ELEMENT CommandMappings (CommandMapping+) >
```

Outgoing Messages

The outgoing messages will always be the same type, because making the type configurable would greatly complicate the code. The configurable header & data should give us all the flexibility we need. The type will be JMS ObjectMessage, where the body is an ActionCommanderMessage object. The ActionCommanderMessage class will be a simple immutable class containing two members: a string containing the data read from the configuration repository, and the DetectionEvent that has been passed along from the Event Source Manager (we might need it, & we've got it, so why not).

Action Component

Action Components are custom components in the TRCF. They are the business-specific operations carried out in response to a detection event. Up to this point in the Framework, all the (standard) logic about which actions to carry out has been applied, and it is now up to the individual Action Components to perform their task. Because there are no limitations placed on what an Action Component may do, we will not try to restrict the developer from any implementation. However, there is one fact in the architecture that dictates how each Action Component must begin its implementation. This fact is that all commands to an Action Component reside in a JMS Topic called the Command Topic.

Action Components therefore will be implemented as Message Driven Beans (MDB), which pull messages off the command topic, and execute their preprogrammed functionality. The functionality carried out by the action component could be done directly in the event handler for the MDB, or the MDB could call a worker bean that performs either a single task, or several tasks by implementing a session facade to call any number of business objects necessary. This document provides the developer with a set of Action Components, which provide a set of "best practice" examples.

Several examples of an Action Component are listed below

Action Device	**Action Component**
Music storage	Deliver music file
WebCam	Take Picture
Printer	Print Coupon
Pager	Send Alert
Database	Locate Target
TurnStyle	Open Turnstyle

Note that an Action Component is identified by a verb, while the potential Action Device (which carries out the task) is typically a noun. Also note that not every Action Component has a physical, external device associated with it. In the case of Locate Target, the MDB / worker bean merely updates the internal database, indicating where the current target has appeared based on the detector they passed by. By default, each supported Action Device has an Action Component associated, which only performs the device's single action. Action Components, which access several Action Devices, to carry out a complex series of tasks implement the session facade pattern using a stateless session worker bean called from the MDB.

The Action Component is related to the system through the JMS Topic it pulls messages from, and the Action Device Manager (if needed) that the Action Component calls to interface specific hardware for executing an operation. These relationships form the basis of many design considerations.

1) What JMS Command Topic does the Action Component watch?

Message Driven Beans can only watch a single topic, and this is assigned with an application-server-specific deployment descriptor tag. Administration UI tools may be used to associate an Action Component with a topic. Initially, all Action Components will watch the same topic, and use message filters based on the Action Component's name to pull only those messages destined for the specific Action Component.

2) How are transactions employed in Action Components?

JMS transactions will not generally be employed. Action Components typically do not rely on the presence of several related messages to perform a task (although its not out of the question). Transactions used by an Action Component might involve database transactions – because of updating several rows of information. Connectors to external systems may also require a transaction. The needs are usually custom for every ~~Action Component.~~

3) What message formats does an Action Component send?

While received messages are always a serialized DetectionEvent object, ~~the sent message~~ (if any) is entirely dependant on the device that Action Commander communicates with. Some actions will use objects that access the framework's internal database. Other objects will construct an XML message to transmit information to other systems. Once again, the needs are custom per Action Component.

4) What kind of unit and system tests can we perform on an Action Component, especially since everyone is custom?

Each Action Component must implement the handling of ~~a test message so that either~~ individual unit tests, executed via specific targets in an Ant build file, or system tests, executed by pumping in events at the front of the framework will produce test results / ~~metrics in a log file~~ which can be examined for any failures.

5) When might we decide to create separate topics for Action Components?

Two possible scenarios include geographic constraints, and when the message filtering mechanism proves inadequate, or inefficient for delivering messages.

6) What concerns exist for deploying Action Components in a clustered app server environment?

Just the usual concerns – they must be stateless.

More detailed information about Action Components and their implementation can be found by examining the javadocs included with the source code. Additional information, such as a deployment descriptor sample is included below.

Sample MDB Deployment Descriptors

<?xml version="1.0"?>

```
<!DOCTYPE ejb-jar>
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>LoggerMDB</ejb-name>
      <ejb-class>com.t_rasa.trcf.actioncomponent.mdb.LoggerMDB</ejb-class>
      <message-selector></message-selector>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
        <subscription-durability>NonDurable</subscription-durability>
      </message-driven-destination>
    </message-driven>
    <message-driven>
      <ejb-name>ListenerMDB</ejb-name>
      <ejb-class>com.t_rasa.trcf.actioncomponent.mdb.ListenerMDB</ejb-class>
      <message-selector></message-selector>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
        <description>Worker Home</description>
        <ejb-ref-name>ejb/worker</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <ejb-link>Worker</ejb-link>
        <home>com.t_rasa.trcf.actioncomponent.ssb.Worker.WorkerHome</home>
        <remote>com.t_rasa.trcf.actioncomponent.ssb.Worker.Worker</remote>
      </ejb-ref>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
        <subscription-durability>NonDurable</subscription-durability>
      </message-driven-destination>
    </message-driven>
    <session>
      <description>Worker Bean</description>
      <display-name>Worker</display-name>
```

```
      <ejb-name>Worker</ejb-name>
      <home>com.t_rasa.trcf.actioncomponent.ssb.Worker.WorkerHome</home>
      <remote>com.t_rasa.trcf.actioncomponent.ssb.Worker.Worker</remote>
      <ejb-class>com.t_rasa.trcf.actioncomponent.ssb.Worker.WorkerBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SimpleMDB</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>Worker</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

JBoss-specific deployment descriptor:

```
<?xml version="1.0"?>
<jboss>
  <enterprise-beans>
  <message-driven>
    <ejb-name>LoggerMDB</ejb-name>
    <configuration-name></configuration-name>
```

```
      <destination-jndi-name>topic/CommandTopic</destination-jndi-name>
    </message-driven>
    <message-driven>
      <ejb-name>RecorderMDB</ejb-name>
      <configuration-name></configuration-name>
      <destination-jndi-name>topic/CommandTopic</destination-jndi-name>
    </message-driven>
    <secure>false</secure>
    </enterprise-beans>
</jboss>
```

Database Design

The Tabula Rasa Component Framework contains its own database with a set of tables. Additional tables are typically required to realize a specific installation of the TRCF. There are 2 standard tables – Target and Event. The Target table is used to define all the unique identifiers that can be detected. This could consist of RFID tag id numbers, or bar code serial numbers, whatever the customer desires. The Event table is the record of all detection events.

Additional tables are required for specific business requirements. For example, a music or video content access system may assign target ID's to specific movies, or other video content, or to specific music performances. When the relevant tokens are presented detected by the system, programs might be used to view details about a piece of content matching the detection event. *This requires an association between metadata about the content and a target id. The* data would be stored in its own table, with a reference to a specific target id in the target table. Sometimes the data may reside in a system outside the TRCF. The target table might have a direct association with a primary key value in another database in another system. In this situation, a custom application would discover the primary key though the event and target information in the TRCF, and then query an external system for the remaining data.

Each table used in the TRCF will have specific administration requirements, which impact the administration UI design. Refer to the section on administration for details on extending the administration UI for your venue-specific requirements.

Standard SQL scripts will be developed that allow the database to be created dynamically. This will allow rapid build and deployment of systems. Test suites can include this step to ensure that this phase of the installation and deployment operation works consistently.

In addition to database creation, there is also the matter of populating table. Typical database population methodologies include:

- One time importing for specific events
- Scheduled importing of target data from external systems
- Manual database entry
- External reference of target data in real time

Entity Relationship Diagram

Standard

The following diagram depicts the base TRCF database layout. The general features realized with this design are:

- Manage a list of valid Targets, Detectors and Operatives
- Record detection events in an Event table
- Optionally associate Operatives with specific Targets
- Refer to an external system for detailed information about a specific person or object

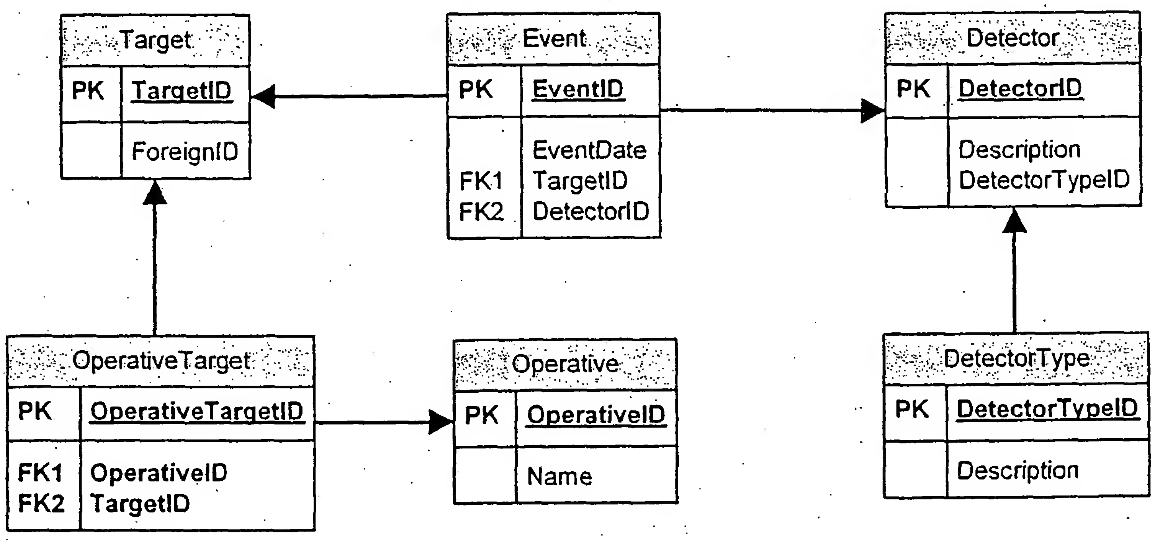
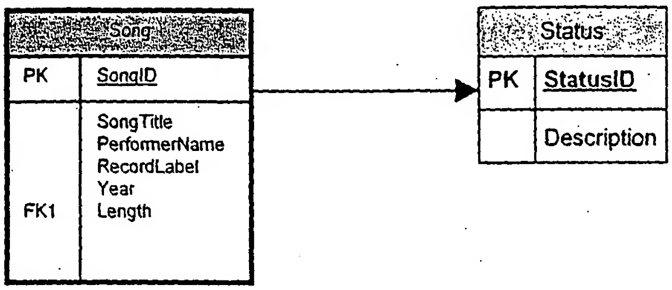


Fig. 11

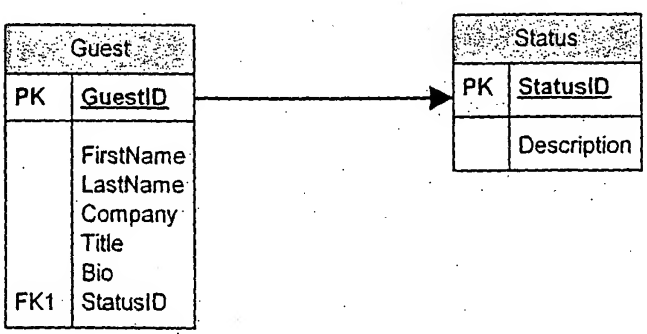
Sample Music Content Retrieval

The following diagram depicts additional tables necessary for a specific implementation of the TRCF. The SongID in the Song table is referenced in the Target table's ForeignID column to associate a specific song with a specific identifier. In this example, data about individual songs resides within the system's internal database.



Sample Casino Application

The following diagram depicts a venue specific implementation of the TRCF where there is reference to an external database. The Target table's ForeignID column associates a specific guest in an existing CRM system (for example) with a specific identifier. User interface applications may employ Expresso's MultiDBObject for tying together detected targets with their associated information.



Data Access

Data access in the TRCF will be accomplished via JDBC access from stateless session beans. The decision to avoid entity beans was made to avoid performance issues surrounding entity beans. Data retrieved from the database will be serialized as XML for consumption by the various application components. Refer to the Component Design section of the document for more information on database access in the TRCF.

Data Dictionary

target

The target table is the repository for all defined targets in the system. In the case of RFID tag usage, there would be one row for every individual tag

targetid – varchar(64) – The serial number recorded in an RFID tag for example. This could also represent a barcode number, or any other unique identifier that can be detected.

guestid – varchar(64) – A foreign key reference to either a venue-specific table within the TRCF, or to an external database system. This reference allows applications to obtain detailed data about detected individuals.

event

The event table contains all detection events recorded by the system.

eventid – int – the primary key; an incrementing number

ventdate – datetime – the system time when the message was received

deviceid – int – a foreign key reference to a row in the Device table

targetid – varchar(64) – a foreign key reference to a row in the Target table

insertdate – timestamp – when the record was inserted in the database

device

A device refers to any managed device within the system. For detection devices, this means...

deviceid – int – the primary key; an incrementing number

d scription – varchar(255) – a unique description for a specific device such as "Left Door".

devicetypeid – int – a foreign key reference to a row in the devicetype table

devicetype

The device type allows for a grouping of detection devices into a common unit. Messages received from a specific device type will result in specific business logic executing based on mappings in the Action Commander configuration file.

devicetypeid – int – the primary key; an incrementing number

description – varchar(255) – a unique description for a specific detector type such as Ballroom. This description is meant to convey a collection of related detectors. In the example given, the Ballroom could have several detectors.

operative

Operatives are the users of a TRCF system. They login to access system features and data. Initially there are three types of users: System Administrators, Venue Administrators, and Hosts.

A single logon each for System and Venue Administrators will exist. Flexible definition of Hosts and their passwords will allow the system control over those individuals trying to access the business-specific application data

operativeid – int - the primary key; an incrementing number

username – varchar(64) – the login user's login name

fullname – varchar(64) – the login user's full name

password – varchar(64) – the login user's password

operativetarget

One of the fundamental features in the TRCF, is that Hosts receive alerts when specific guests are detected. This table allows for the association between hosts and guests. Administrative interfaces will allow editing in this table.

operativetargetid – int - the primary key; an incrementing number

operativeid – int – a foreign key reference to a row in the Expresso user database. Each operative defined in the system will have login functionality in the Expresso-based administration and user applications. All details about the operative will come from this association.

targetid – varchar(64) – a foreign key reference to a row in the Target table

guest

Guests, and the data that is captured for them is one of the variable portions of the system. Each business will capture different data for consumption by Hosts when executing their application.

guestid – int – the primary key; and incrementing number

firstname – varchar(64)

lastname – varchar(64)

Administrative Tools

The Administration system is a web application built upon the Jakarta Struts Framework. The framework allows for rapid development of data-driven web applications. This is accomplished through the use of standard, tested components for the presentation, business logic, and database layers of the architecture. The Administration UI tools are divided into several categories, depending on the user's role, and the task being performed.

General Access Control

Administrators define who can access both the administrative and venue-specific applications. This will follow a role-based security scheme. The general roles in the system are

- System Administrators – Systems ship with a single user defined as an administrator. This user has a default password that should be changed when the system is initially installed. Can define detector types, detectors and assign actions to types
- Venue Administrators – Manages data specific to the venue, such as the list of targets
- Operatives – can access venue-specific applications & data

System Definition

System Administrators can access this section of the administration system to define details specific to the deployment.

Defining Equipment & Component Deployment

Define Detector Types – logical groupings of detectors with similar purpose, like "Front Doors"

Define Detectors - A physical device or system that can send detection events to the system. Each detector has a type and description. The ID is automatically assigned by the system.

Associate Detector Types and Actions – A process that links events from common detectors (like "Front Doors") with a set of specific actions (supplied by Action Components deployed with the system)

Future:

Define Detectors – additional administration functions will be added to allow remote configuration and monitoring of detectors.

Define Multiple EventTopics - A collection point for detection events from specific detector types. Initially, there will only exist one topic for all events called the "EventTopic", but in the future, each detector type might be assigned its own topic. If requirements demand, we may want to dynamically create event topics, and assign them the output of various groups of detectors having logical associations not achieved via the topic-per-type method.

Note: You want to collect detection events from multiple detectors into one location if they mean the same conceptual thing. For example, three entrance door detectors all have the same meaning (type) - a target has walked into (or out of) an area. These events should all collect in one place. This does not mean the events are not distinct. Each event has a message that contains the identifier of a detector. If specific business actions dictate, we could still choose to perform different actions based simply upon the detector identifier

Add Detector Types to Event Topics – If requirements demand a scheme other than one event topic, or one event topic per detector type, we may need a user interface for dynamically assigning detector types to event topics

Define Multiple Command Topics - A collection point for action commands from the action commander. Initially there will only be one command topic, and action components will filter for their specific commands using the message-filter mechanism built into message-driven beans.

Add Action Components to Command Topics - The association between an action component and one or more Command Topics – this will affect the deployment descriptors for the message-driven beans that are the front-end to all action components. This is only needed if defining multiple command topics.

Define Business Rule - A named business objective

Define Logic for Rule - Determines target Command Topic destination(s) based on database and/or message data

Define or import new device managers – a way to dynamically deploy new detector device types (as they become available) into an existing system.

Define or import new action components – a way to dynamically deploy new action components (as they become available) into an existing system.

Screens:

Target data import (with dynamic column definition capabilities)

Target data editing (like assigning an ID to the target)

Status type editing (what status can I assign the user during the event)

Device Manager (detector) definition

Action Component definition

Operative definition

Business Rule Editor (map events to commands)

Target - Operative association (specify a specific operative to handle a target)

Detector - Operative association (an operative is notified of all detections at specific location(s))

Camera - Detector association (which camera does the detector use to obtain a picture)

Database backup / restore / export screen

Event Management (purge)

MEM specific issues:

Follow-up email template editor

Follow-up email transmission screen

Report selector / viewer

Report editor screen

System Usage

This section of the administration application involves venue specific tasks such as defining targets, and making assignments between operatives / roles and targets.

Database Administration

Database administration involves creating the database and appropriate tables. Where possible, data supplied by outside systems can be used to automatically configure the appropriate tables and columns

Importing Data

Local target data elements are database columns that represent information about the target clients. They are local because they are captured/imported into the local Tabula Rasa database.

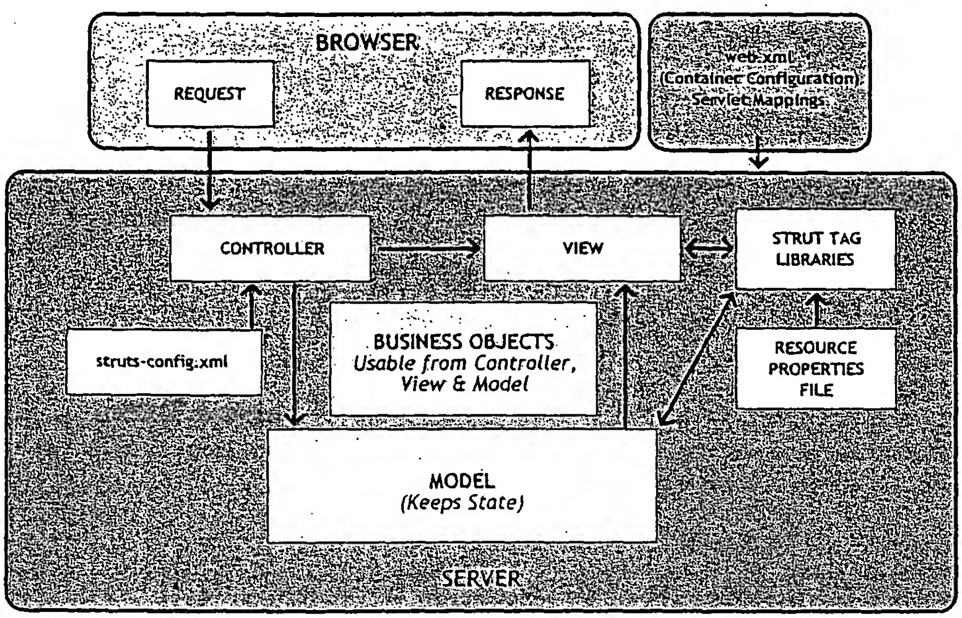
Exporting Data

Exported data will consist of collected target and tag data, as well as any data generated through active use of the system, such as detection events that are persisted in the database. This information will be used for reporting, or as input to other systems.

Editing Records - add delete update

A user interface allows administrators to add, edit, and delete any table data from the database. The most likely scenario is the manual addition and deletion of targets from the mandatory target database table.

Client Applications



Client programs (which include administration) are web applications based on the Struts Framework. In short, this dictates the design of all applications in the TRCF. Refer to the Struts documentation for further details on its Model 2 (MVC) based application design principals.

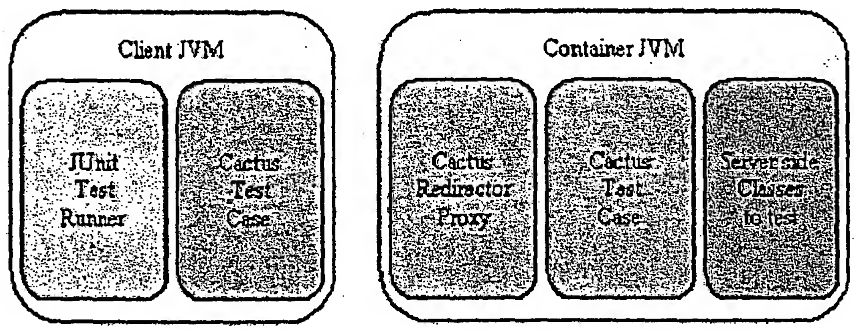
Other resources on the Internet provide great a great deal of information about designs based on Struts:

http://www.pcci.ch/pdf/j-struts.pdf

This document contains class diagrams and detailed analysis of Struts design.

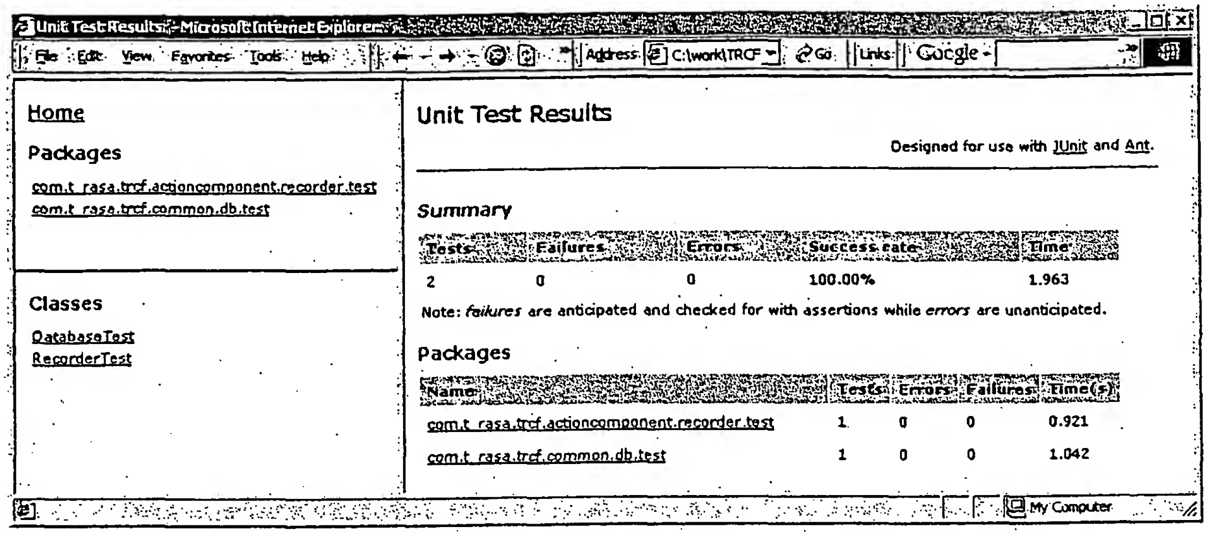
Testing

The TRCF uses JUnit and Cactus for its testing mechanism. JUnit is a simple framework to write repeatable tests and is geared towards testing individual classes. Cactus is an extension to JUnit that allows for tests to run within the virtual machine (VM) of the application server. It is geared towards testing server-side java code (Servlets, EJBs, JSPs, TagLibs). There are certain situations where JUnit tests will be employed to test (EJBs for example) when it is impractical to run a test within the VM of the application server. This is primarily due to the disconnected, asynchronous nature of the application, and the need for fine grain thread control not allowed inside of an application server environment. The following diagram illustrates the Cactus view of testing.



Tests will be initiated via Ant build targets and will allow testing of individual components of the framework, as well as perform overall system tests. Testing will fall into 3 general categories. Unit tests allow the developer to provide various inputs to a system component and test for the anticipated result. System tests allow the developer to provide inputs to the system and see how the individual components function together to process those inputs. Load and Performance tests allow the developer to run tests repeatedly, simulating various levels of simultaneous inputs.

HTML formatted reports are generated for all tests. The format used is the default JUnit style.



Unit Testing

Each component of the TRCF will have a test folder that contains the classes needed to conduct individual component tests. The tests for one component must not rely on other components in the system. The following components have tests defined:

Event Processing Engine:

Device Manager

Event Source Manager

Event (JMS) Topic (all TRCF-defined topics)

Monitor

Action Commander

Command (JMS) topic (all TRCF-defined topics)

Action Components (all message-driven bean components in the library)

Action Components support classes

Refer to the individual JavaDocs for each test component for details on the individual tests defined.

Applications:

System Testing

System testing exercises all the components from the unit tests yet does so by relying on the interdependencies of all the components. For example, to test the event-processing engine, the

system tests need only cause device managers to generate detection events. Those with valid formats will be processed all the way through to Action Components. Note that the device manager must generate the following events:

Valid format messages

Valid format messages with invalid message contents – such as an invalid detector type

Load / Performance Testing

There are three areas that can provide load to the TRCF system:

1) Operating System tasks / processes
2) Event Processing Engine
3) Administrative & User Applications

Tests are constructed that let the user affect 2 and 3 listed above. In general, these tests are extensions of the system tests outlined in the previous section. Instead of running single passes of system tests, and observing the output, we run many tests simultaneously, simulating many detectors and many application users. The ratios of detection events and client accesses are adjusted to approximate real-world access to the system. The metrics tracked include:

1) Total events created / processed
2) Events processed / second
3) Average event processing time
4) Average web application page access time.

Source Code Control

Stellcom uses Microsoft's Visual Source Safe to maintain software revision history for the TRCF. The following general organization of projects will be employed to maintain and track versions of the TRCF and venue-specific implementations. The following assumptions are made about the future development of code for Tabula Rasa:

1) There will be a single, core TRCF collection of versioned code. In general, this includes the event processing engine, a collection of standard action components (library), the core administration user interface and required components, and a base user interface for viewing events processed by the engine. Any enhancements to the TRCF must be applicable to any venue, and in general, increase the intellectual property of Tabula Rasa by creating new "libraries" of functionality.
2) There will be several standard vertical market implementations of the TRCF. These implementations will be standard deployments that can be delivered as-is to individual clients, or serve as the basis for modification for clients that need additional functionality above and beyond the base offering
3) Venue & Client specific implementations include those modifications to the standard vertical market implementations that are useful only to a given client, and do not build additional desired functionality into standard offerings.

Given the above considerations, and the features provided by Source Safe, the following layout and procedures will be followed to ensure the maintainability of software releases.

Layout

TRCF – The base Tabula Rasa Component Framework

```
Dev
  Cpp             c++ programming language projects
  HTML            static web content
  Images          web site images
  Java            all java source files
    com
      t_rasa
        trcf
          ... indvidual components ...
  JSP             dynamic web pages
  META-INF        EJB definitions
  SQL             schema files for initializing the database
  WEB-INF         web site definition & support files
  XML             configuration files
Doc
```

Architecture

Design

Install

Device	for constructing the device installation
Jboss	JBOSS-specific installation files
Weblogic	Weblogic-specific installation files
Test	test configuration files and data
UEE	deliverables from UEE

SupportSoftware

MEM – A base, vertical-specific product

Dev

Java — all java source files

com

t_rasa

mem

... indvidual components ...

Versioning

Versions will be maintained at the TRCF, vertical, and venue-specific levels in Source Safe. For a given release of these projects a label will be applied indicating the version level of the release, using a major_release.minor_release.build (xx.xx.xx) format system. Release notes for the TRCF will specify the changes made from the previous release, known bugs, deployment instructions, etc. The vertical and venue-specific project releases will contain this same level of detail, but will also specify the TRCF version required by the release. To reconstruct a system, the developer need only get the version of TRCF required by the specific implementation to build, test, and repair bugs.

Branching

If a developer must make a change to a vertical or venue-specific release that requires a change to the TRCF, and the version of TRCF required is older than the current release, the developer must either bring the vertical or venue-specific release current with the latest TRCF, or branch the TRCF code so that TRCF development can proceed on separate paths, until such time as the code can be merged into the TRCF, or the vertical / venue release can be made current with the latest TRCF.

These notes supplement standard Stellcom Visual Source Safe Configuration Management documents and are only meant to provide details specific to Tabula Rasa.

Deployment Details

Supporting Software

The following software versions are used in the TRCF and can be located at the sites indicated.

Java Runtime and SDK

Java SDK 1.3.1 – http://java.sun.com/j2se/1.3/

Linux & Windows copies maintained in

$/TRCF/SupportSoftware/java/ j2sdk-1_3_1_01-win.exe

$/TRCF/SupportSoftware/java/ j2sdk-1_3_1_01-linux-i386-rpm.bin

Solaris version not in Source Safe – visit java.sun.com to download if required

WebLogic Application Server

BEA Weblogic 6.1 - Maintained by Cindy Alford for Stellcom

Jboss Application Server

Jboss 2.4.4 / Tomcat 4.0.1 – http://www.jboss.org/binary.jsp

$/TRCF/SupportSoftware/jboss/JBoss-2.4.4_Tomcat-4.0.1.zip

MySQL Database & Driver

MySQL Database Server 3.23.47 – http://www.mysql.com

$/TRCF/SupportSoftware/mysql/MySQL-3.23.47-1.i386.rpm

$/TRCF/SupportSoftware/mysql/MySQL-Max-3.23.47-1.i386.rpm

JDBC Driver mm.mysql-2.0.8 – http://sourceforge.net/projects/mmmysql/

$/TRCF/SupportSoftware/mysql/mm.mysql-2.0.8-bin.jar

Oracle Database

Oracle 9.x - Maintained by Cindy Alford for Stellcom

Jakarta Struts

Jakarta Struts 1.0.1 – http://jakarta.apache.org/builds/jakarta-struts/release/v1.0.2/

$/TRCF/SupportSoftware/struts/jakarta-struts-1.0.2.zip

WebCam Software

Many webcam software packages were sampled during the course of the prototype. The one that seemed to perform the best was WebCam32 from Surveyor Corporation.

http://www.surveyorcorp.com/products/webcam32.html

$/TRCF/SupportSoftware/webcam/webcam32.exe

Log4j

Log4j 1.1.3 – http://jakarta.apache.org/log4j/docs/download.html

$/TRCF/SupportSoftware/log4j/jakarta-log4j-1.1.3.zip

Note that many packages include a version of log4j. This is meant for those installations that do not have log4j.

JUnit & Cactus w/Ant

Cactus 1.2 - http://jakarta.apache.org/builds/jakarta-cactus/release/v1.2/

$/TRCF/SupportSoftware/cactus/jakarta-cactus-ant-1.4-20010925.zip

Note that the release of cactus comes with a junit, cactus and ant integrated. By expanding the zip, all the necessary libraries will be included. You will need to copy junit.jar and cactus.jar to your project and deploy them with your code. Versions are currently deployed with the TRCF build and can be obtained by examining out the latest code. You are also required to download the optional jar and place it in the $ANT_HOME/lib directory.

http://jakarta.apache.org/builds/jakarta-ant/release/v1.4.1/bin/jakarta-ant-1.4.1-optional.jar

Configuration

Low-End Configuration

(1) Combined Web / Application / Database Server

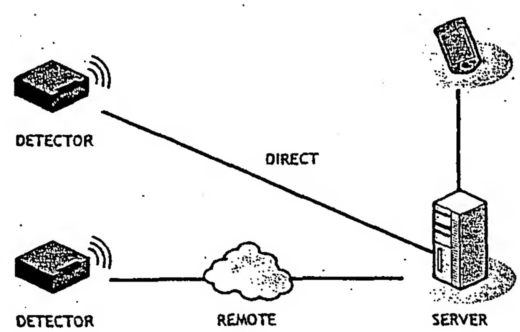


Fig. 12

High End Configuration

(2) Web Servers w/Load Balancer

(2) Clustered Application Servers

(2) Clustered Database Server

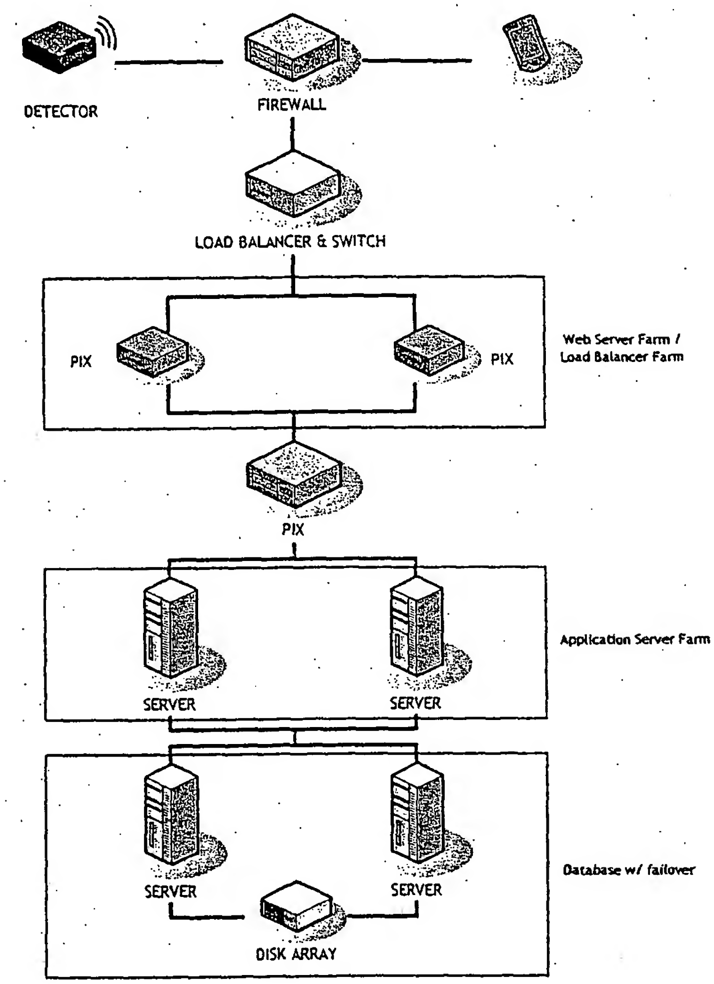


Fig. 13

Engineering Perspectives

While timing has always been a critical factor to adopting new technologies, the accelerated pace inherent in a virtual, information-driven business model has put even greater emphasis on response times. Tabula Rasa is no exception. It is imperative not only to project enterprise systems into various client channels and to do so repeatedly and in a timely manner, with frequent updates to both information and services. The principal challenge is therefore one of keeping up with the hyper-competitive market while maintaining and leveraging the value of existing business systems. In this environment, timeliness is absolutely critical in gaining and maintaining a competitive edge. A number of factors can enhance or impede Tabula Rasa's ability to deliver custom applications quickly, and to maximize their value over their lifetime. The engineering perspectives presented here provide a baseline by which the TCRF system architecture has been designed.

Client-Side Issues

The component framework toolkit includes a client side application foundation that provides fundamental services in a secure and rapidly deployable environment. These services include:

- **Security:** This service ensures that that all users are properly authenticated and that all business data transferred between the device and back-end servers is encrypted and secure. Authentication can be handled through a simple user ID and password or through a more secure, digital certificate based mechanism The framework gives Tabula Rasa the ability to offer, at a cost, appropriate security to address individual customer needs.
- **User Session Management:** The client and server cooperate to ensure each user is assigned a unique session that allows the system to associate session specific information with the user for the duration of the business transactions. This session must remain valid through intermittent wireless network connections.
- **Miscellaneous Issues:** There are many other client side issues that need to be taken into account when building a solution. These include the proper management of scarce device resources, component framework version control, device provisioning and more.

Server-Side Issues

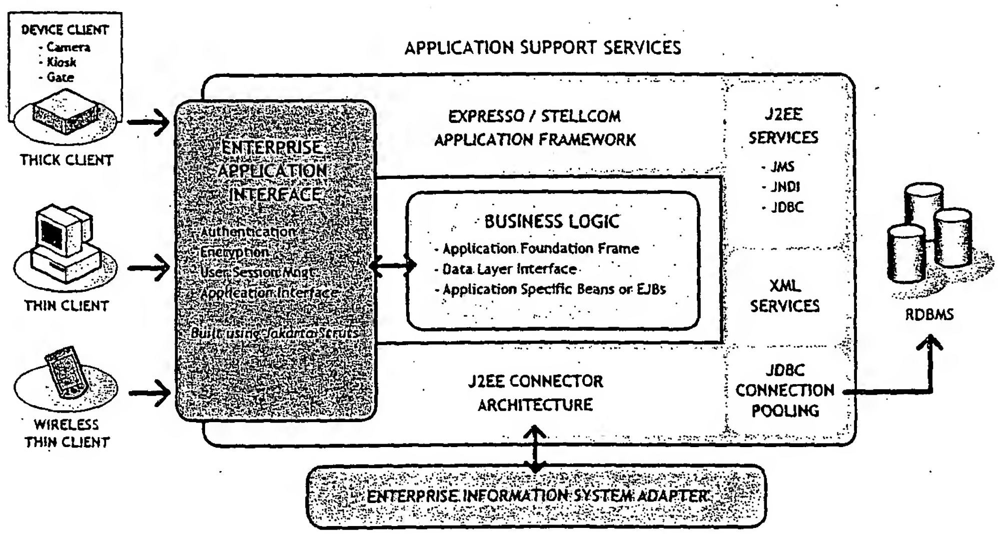
A component framework's server side architecture faces the following challenges:

- The architecture must fit cleanly into any existing IT infrastructure and place a minimal burden on existing customer IT resources.
- The architecture must support the needs of the client application (e.g. security, session management, etc).
- The architecture must offer clean and simple interfaces to all application
- The architecture must be easily extensible to accommodate additional devices.
- The architecture, and its interfaces, must be tight, clean, simple and very well documented allowing developers to rapidly build custom component frameworks.

- The architecture must offer an application frame that can be quickly extended and customized to specific client application needs.
- The entire component framework product should be sensitive to customer technology preferences.

Java / J2EE

A Java / J2EE server-side solution offers a platform independent, standards based component framework. The J2EE Component Framework presents a minimal impact on any existing IT infrastructure. It is comprised of a proven set of components that collectively make a solid technical foundation for any business application. At the core of this solution lies an application skeleton built around a set of proven design patterns. This application skeleton offers a pre-packaged starting point supporting the rapid development of custom components. Users simply extend the various interfaces and implementations to build their custom applications.



The application skeleton leverages the services available from the Stellcom / Expresso application framework. This framework includes a variety of XML processing tools, various helper beans and EJBs, Object / Relational database mapping tools and more. These tools couple with the standard J2EE development technologies to form a powerful application development environment

Interfacing to third party enterprise applications is handled through the industry standard J2EE CA (Connector Architecture). The J2EE CA provides Tabula Rasa with the ability to quickly integrate with commercial enterprise applications. J2EE CA adapters are available off the shelf

and provide a standards based means to access enterprise applications, reducing both development cost and time. There are currently numerous J2EE CA adapters available to wide variety of enterprise applications and data sources (e.g. Baan, JD Edwards, SAP R/3, PeopleSoft, Siebel, Vantive, CICS, CORBA, MQ Series, etc). These adapters are available from a variety of vendors (e.g. Insevo, Actional Corp, Infogain, AMS, Attunity and others). The list of supported enterprise applications is rapidly growing ensuring clean and easy access to corporate enterprise data. Where applicable, the framework provides a thin abstraction layer isolating users from the details of J2EE CA adapter interfaces.

The server-side framework provides a package of objects supporting communications and transactions with the client applications. This package provides all the server-side functionality supporting transactional operations, authentication and encryption, user session management and more. Users design their custom component frameworks to interact with these built-in services through a series of public APIs. An API, or application public interface provides a standard definition for accepting and sending data.

A Word About J2EE

J2EE is the chosen architecture platform for the TRCF. J2EE offers the following advantages:

- J2EE is pure Java from the presentation layer (Java Server Pages/JSP, Java Foundation Class/JFC, also known as Swing), through the middle tier (Enterprise Java Beans/EJB, Java Transaction Server/JTS, Java Messaging Server/JMS, etc) to the backend database access (Java Data Base Connector / JDBC). A single language development environment adds cohesiveness to a system design and allows developers to concentrate on developing the system rather than on learning proprietary development methodologies.
- As a pure Java based solution, J2EE based systems are fully portable to any platform (NT, Solaris, Linux, etc).
- Unlike proprietary solutions, any application written to the J2EE standard can be directly ported to any other J2EE compliant application server. Applications are not tied to the success of one vendor.
- J2EE solutions are fully scalable. Application software scales cleanly as additional EJB components are added to accommodate additional functionality. Additionally, J2EE application servers scale by clustering while load balancing EJB component usage and handling EJB fail-over transparently.
- J2EE applications offer enterprise level transaction handling. Business transactions can span any number of EJB components and can be committed or rolled back at any point in the transaction.
- J2EE has been embraced as the development standard by all the major players in the advanced technology space. As an industry standard, J2EE compliant solutions can leverage a huge array of proven components, products and libraries from companies like IBM, Oracle, BEA, Sun, etc. The availability of components, tools, products and solutions from companies such as these helps speed development and insures J2EE solutions are built on technically proven foundations.

Framework Architecture

This section describes the architecture of the component framework, exploring the partitioning of functionality into components and the decomposition of the components.

Overview of Application Components

The functionality of the system is partitioned into the following components, each of which may have multiple instances:

- **Device Manager:** Handles the physical operation of a device and provides a standard system interface;
- **Event Source Manager:** Receives messages about physical events and packages that information into an appropriate Event Topic;
- **Monitor:** Passes messages from an Event Topic to the appropriate Action Commander;
- **Action Commander:** Creates rudimentary business commands triggered by the physical events; and
- **Action Component:** Communicates with devices and external databases.

Behaviors

The following scenario describes the various behaviors of the system, organized by component. It is provided to illustrate how these components operate in the context of the TRCF.

1. A detector senses a target. The *Device Manager* that communicates with the detector receives the information and translates it into a message the system can interpret. This message may include an ID for the target and an ID for the detector.
2. The *Device Manager* sends the message to the *Event Source Manager*, which creates an Event Message. An Event Message may include additional information, such as the time of the event. The *Event Source Manager* will persist the event message in an *Event Queue*.
3. The *Monitor* listens to the *Event Queue*. When the *Monitor* sees a message, it will query the system database to see if the target ID is recognized. It will also check for business rules that pertain to the target ID. The *Monitor* will append any the target ID information to the message and route it to one or more instances of the *Action Commander*.
4. When the *Action Commander* receives an Event Message, it queries the system database to determine if there are any business rules related to the event. The *Action Commander* then translates the message into one or more Action Commands and persists them in the *Command Queue*.
5. The *Action Component* listens to the command queue. When the *Action Component* sees an *Action Command*, it performs the action. In most cases, it will route the command to an instance of the *Device Manager*, an External Database System, or an Operative. For example, if the action is to take a picture, the *Action Component* will send

a message to the *Device Manager* that controls the camera. The Action Component may also query an External Database System or send a message to an Operative.

6. If the *Action Component* sends a command to a *Device Manager*, the *Device Manager* will receive the command, communicate with the device, and possibly report the status of the device to the system.

Responsibilities

This scenario describes the responsibilities identified for each component. It is provided as a means of identifying the unique responsibilities of each component in the context of the TRCF.

1. The *Device Manager* handles the physical operation of a device and provides a standard system interface.
2. The *Event Source Manager* receives messages about physical events and creates an Event Message, which may contain additional information about the event. It persists Event Messages in an Event Queue.
3. The *Event Monitor* polls the Event Queue, identifies a target ID, determines which instance of the *Action Commander* should be notified, and passes the message from the Event Queue to the appropriate instance of the *Action Commander*.
4. The *Action Commander* receives messages from the Event Monitor, creates rudimentary business commands (Action Commands), and persists those commands in a Command Queue.
5. The *Action Component* polls the command queue and carries out Action Commands. It communicates with Device Managers, queries external databases, and sends messages to Operatives.

Relationships

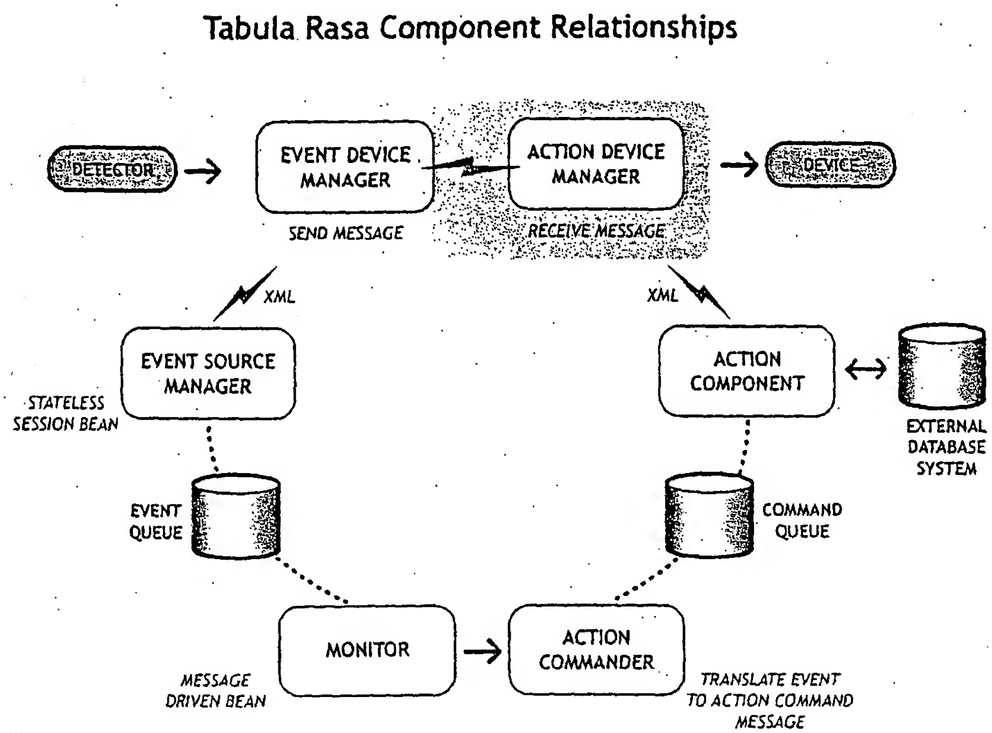


Fig. 14

Design Goals

The TRCF provides a solid base for developing new business processes through the use of several J2EE technologies. Jakarta Struts, is specified for presentation and Web tier development. Expresso supplies a library of general-purpose functionality for the business and data tiers. A set of custom coded J2EE components supplies the functionality to abstract the hardware and physical events from the potential business components.

Business Process Ready

The TRCF will be deployed in conjunction with a set of business components designed to support a specific client or industry's business processes. These business components will be built using the same technologies outlined above, Jakarta Struts, Expresso, and custom coded J2EE components. Because the specific functionally of the business components is not specified in the TRCF, it is designed to support a variety of business components without placing limits on their abilities.

D vice Independence

The TRCF is designed to allow new detection devices or systems to be added transparently to the system so that business components that may make use of them. While the current system is focused on Radio Frequency Identification or RFID technology, any other technology target identification system will fit cleanly in place. This is accomplished by separating physical event processing from the business processing. The use of two different types of message topics, event and business command, isolates not only the identification technology, but also the identification event itself from the business commands that can result.

Quick Deployment

In order to reduce cost and increase speed of deployment, a thin client interface is specified for administration and possible business implementations. A thick client is not precluded from the business implementation. Analysis of the specific business need should determine if a thick client is needed. In either case, a MVC (Model View Controller) pattern, a common industry practice explained in section 3.3, is established for client implementation. Message Flow

The TRCF defines a set of components that can translate physical events into one or more business commands that trigger a business action. Specific knowledge of hardware devices is encapsulated and isolated. Information about an event is marshaled into one or more define event topics. The hardware event (detection) is translated into one or more business commands and placed into one or more business command topics. Business objects subscribe to the topics and listen for commands. The information about business commands is passed to the subscribed business objects for processing.

Model View Controller Architecture / Jarkarta Struts

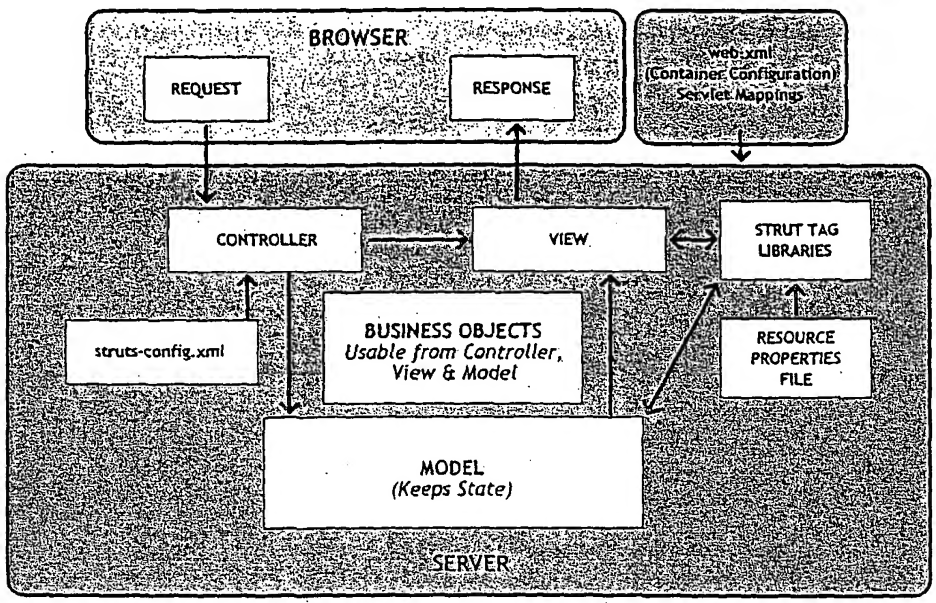
Jakarta Struts is a useful framework for building web applications with Java Servlets and Java Server Pages (JSP Java Servelets and JSP are the J2EE technologies that provide a web interface to an application. It encourages application architectures based on the Model-View-Controller (MVC) design paradigm, colloquially known as Model 2. In Model 2, servlets help with the control-flow, and JSPs focus on writing HTML. True to the Model-View-Controller design pattern, Struts applications have three major components: a servlet "controller", JavaServer pages (the "view"), and the application's business logic (or the "model").

In the MVC design pattern, a central Controller mediates application flow. The Controller delegates requests to an appropriate handler. The handlers are tied to a Model, and they act as an adapter between the request and the Model. The Model represents, or encapsulates, an application's business logic or state. Control is usually then forwarded back through the Controller to the appropriate View. The forwarding can be determined by consulting a set of mappings, usually loaded from a database or configuration file. This provides a loose coupling between the View and Model, which can make an application significantly easier to create and maintain.

Struts includes the following primary areas of functionality:

- A controller servlet that dispatches requests to appropriate Action classes provided by the application developer

- JSP custom tag libraries, and associated support in the controller servlet, that assists developers in creating interactive form-based applications
- Utility classes to support XML parsing, automatic population of JavaBeans properties based on the Java reflection APIs, and internationalization of prompts and messages



Device Manager

The goal of the Device Manager subsystem is to allow the management of physical devices to be isolated from the business logic of the overall system. The classes that manage these devices must contain as little processing as possible, in order to allow them to concentrate on their interaction with the device.

The physical devices fall into two main categories. The first category is the devices that detect the presence of a target, for example RFID readers. The second is the devices that perform some action on behalf of a system, for example a camera.

The first category of devices is managed by subclasses of the DetectorDevice class. These classes generate a DetectorEvent object, and then pass this event off to a processor where it can be delivered to all parties that have registered an interest in it. This handoff allows the DetectorDevices classes to return quickly to reading new events.

The SystemProxy always registers to be informed of events. It has the job of translating the DetectorEvent object into a SOAP, Simple Object Access Protocol, message and sending it to a server process where it will be handled by the business logic. SOAP is specified because it provides a stardards based, technology natural, mechinism for communication. New detection devices, with unforeseen technology, will use the SOAP interface. In addition, category two

devices can optionally register their interest in receiving the event. This behavior supports the ability of a device, such as a camera, to respond to an event quickly, without waiting for the business logic to complete.

Observer / Observable Pattern

The Observer/Observable pattern supports the delivery of events to one or more interested parties. The EventProcessor extends the Java API Observable class, while the SystemProxy and interested Devices classes implement the Observer interface.

Class Diagrams

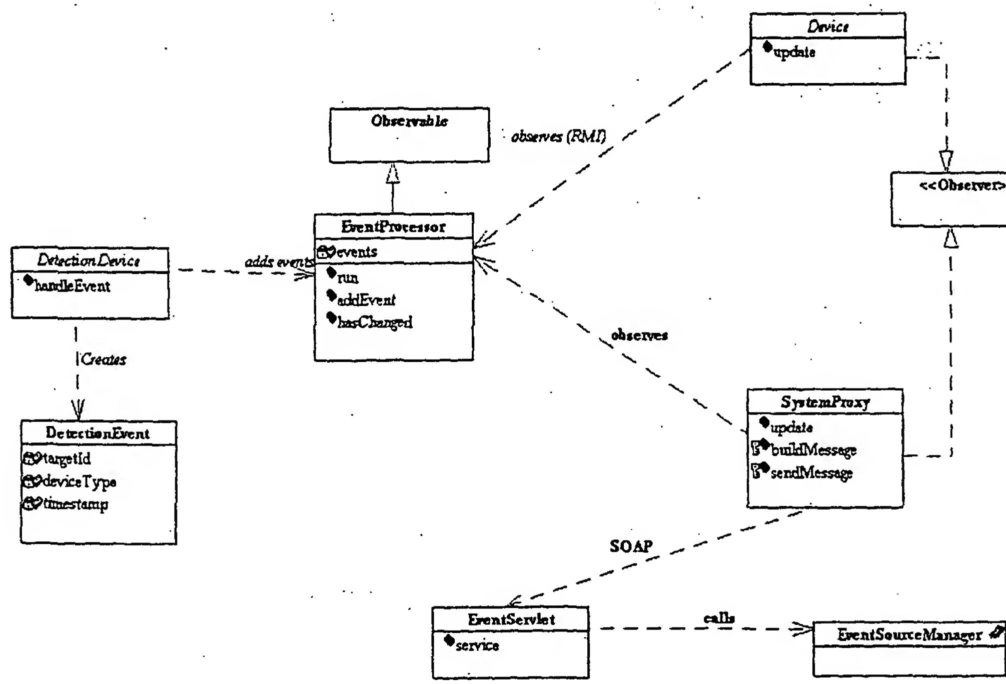


Fig. 15

Sequ nce Diagrams

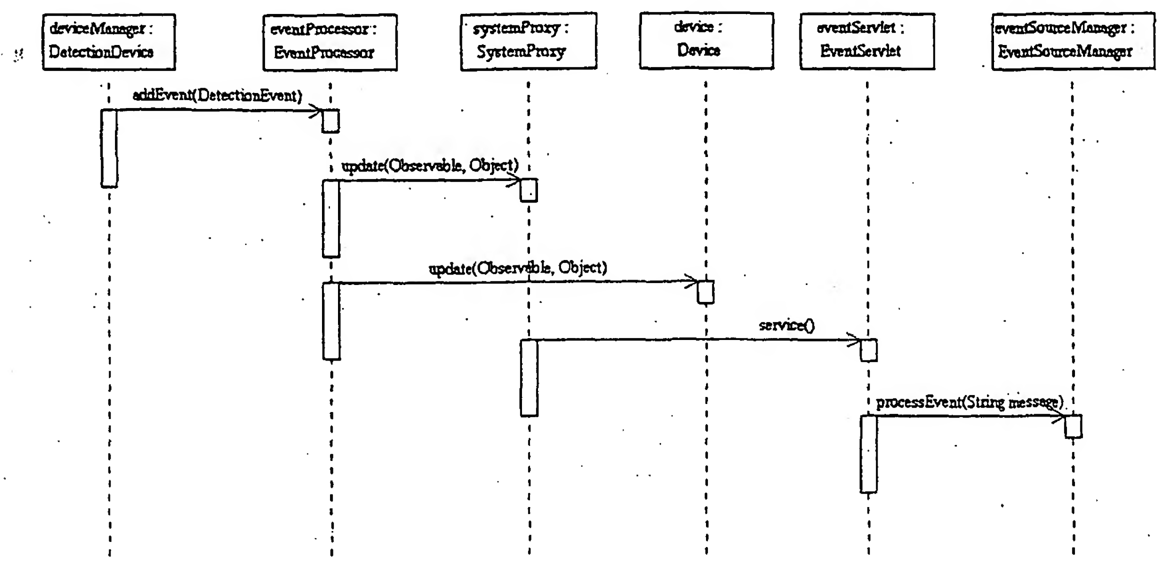


Fig. 16

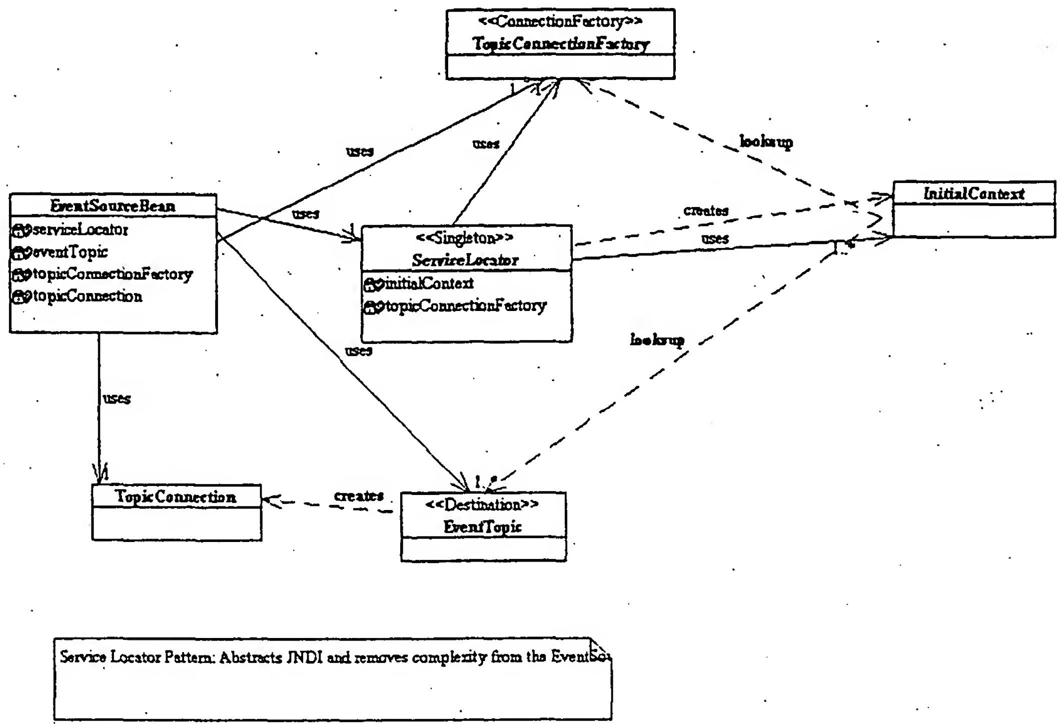
Event Source Manager

The Event Source Manager is tasked with moving information received from the Device Managers and placing the information into an appropriate JMS (Java Message System) topic. The JMS topic lookup and creation involves complex interfaces and network operations. The EventSourceBean is a stateless session bean that has received the information from the device manager. This bean requires access to the appropriate JMS topic to publish the information. A Service Locator strategy is employed here.

The Service Locator for JMS components uses TopicConnectionFactory object in the role of the ServiceFactory. The TopicConnectionFactory is looked up using its JNDI name. The TopicConnectionFactgory can be cached by the ServiceLocator for future use. This avoids repeated JNDI calls to look it up when the client needs it again. The ServiceLocator may otherwise hand over the TopicConnectionFactory to the client. The Client can then use it to obtain a TopiSession or to create a Message and a TopicPublisher.

Service Locator Pattern

The function of the Service Locator is to abstract all JNDI usage and to hide the complexities of initial context create, object lookup, and object re-create. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.



Class Diagrams

Fig. 17

Sequence Diagrams

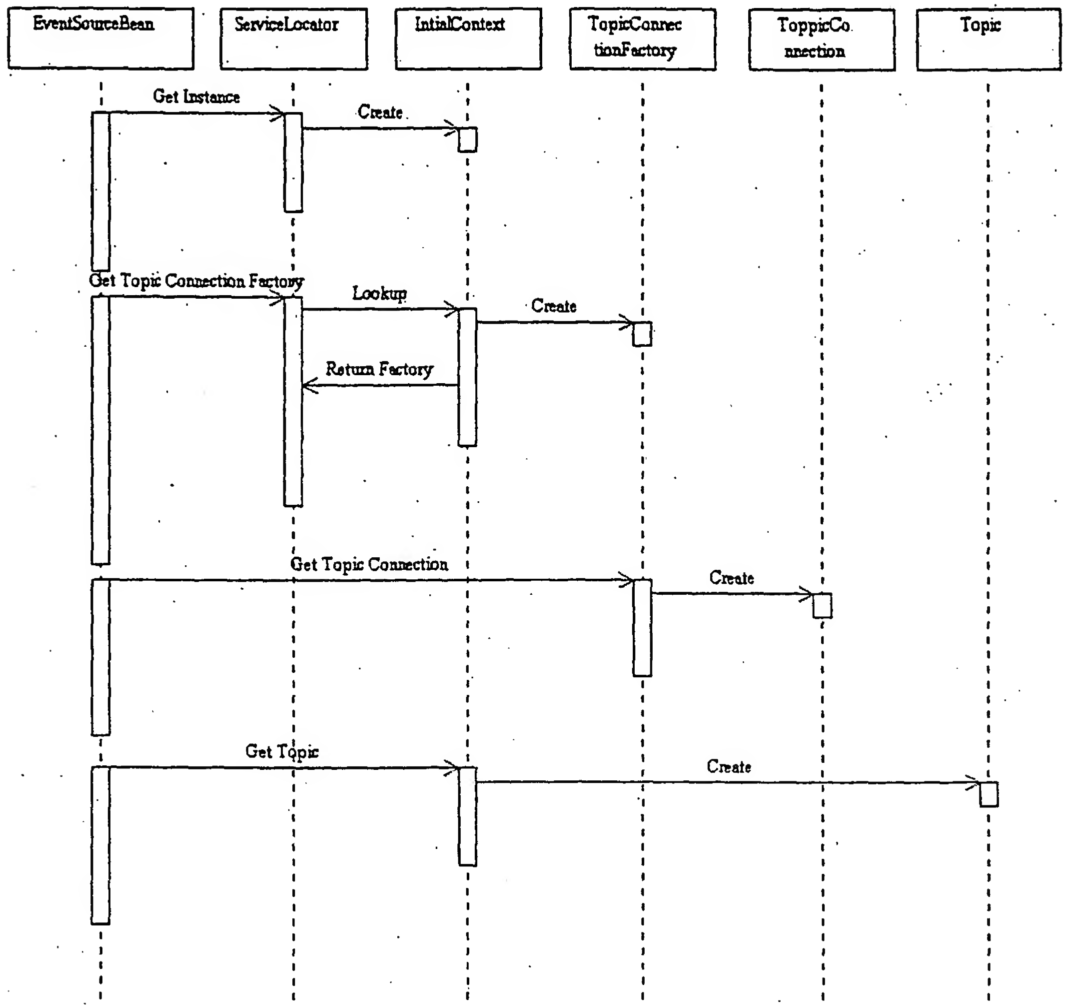


Fig. 18

Monitor and Action Commander

The Monitor System is a series of Message Driven Beans. Message Driven Beans are a standard mechinism within J2EE that have the ability to monitor a message queue and respond asynchronously. Message Drven Beans are employeed here to listen to the various event queues. When a new event message arrives, it is handed off to the Action Commander..No other processing is necessary.

The job of the Action Commander system is to convert system events into business commands and to deliver these commands to the appropriate JMS (Java Message System) command topic. JMS is a stardard interface to the J2EE message oriented middleware. A JMS topic is a queue of messages that can be published to multiple readers, all of which are guaranteed to receive the message before it is removed from the queue. The definition of how events map to series of commands is an administrative task.

The Action Commander contains a collection of event mappings. When it receives an event message from the Monitor system, the Action Commander examines the event type. It then looks up the mapping for this event, which will be a list of commands and the business topics to which each should be delivered. The Action Commander enhances the event message with the name of the command, and then delivers the new message to the appropriate business topic. It uses the ServiceLocator to find and connect to this topic.

Command Pattern

The Command Pattern forwards a request only to a specific object. It encloses the request for a specific action inside an object and gives it a known public interface. It lest you give the client the ability to make requests without knowing anything about the actual action that will be performed and allows you to change that action without affecting the client program in any way.

Class Diagrams

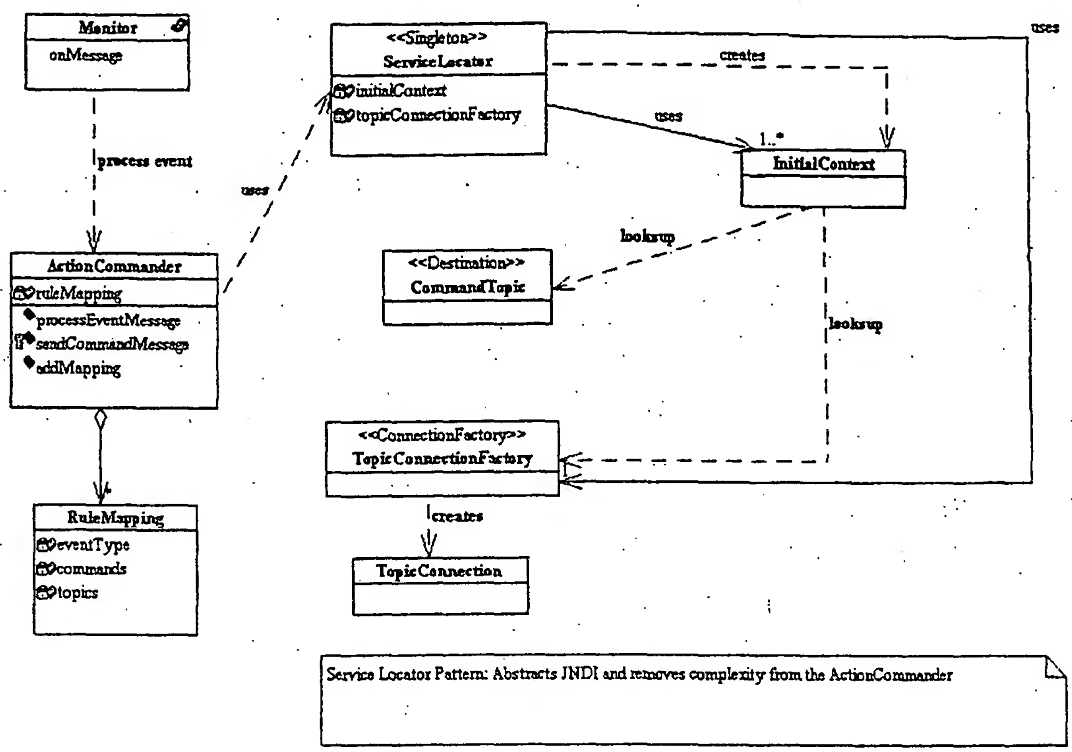


Fig. 19

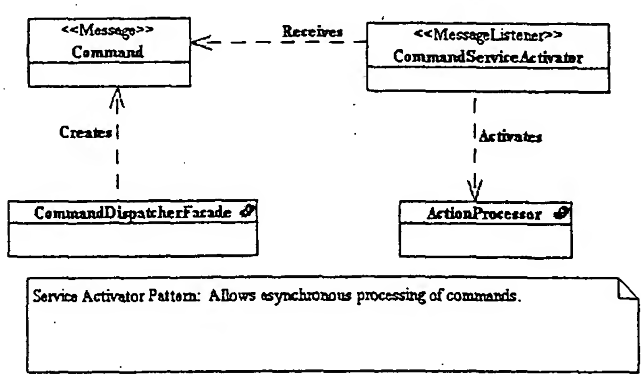


Fig. 20

Sequence Diagrams

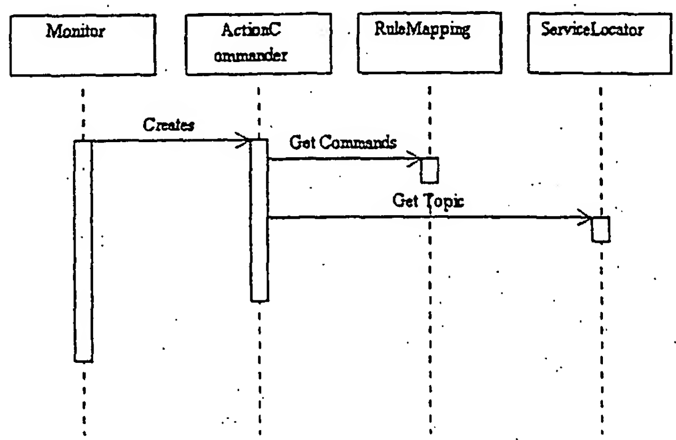


Fig. 21

Action Component

Action Components represent the division between the generic framework of the system and the specialized development for each application. The system framework will handle converting system events into a series of user-defined commands and publishing these commands to a series of JMS topics. Custom action components can subscribe to these topics, and then respond to incoming commands in a system-dependent way.

Several generic Action Components are included with the framework. They represent functions likely to be needed by many systems, as well as the best practices for developing custom components for this framework. They will rely heavily on the Service Activator pattern, see section 3.7.2, for reacting to incoming command messages and the Session Façade pattern, see section 3.7.1, for communicating with the serverThe generic Action Components currently defined are:

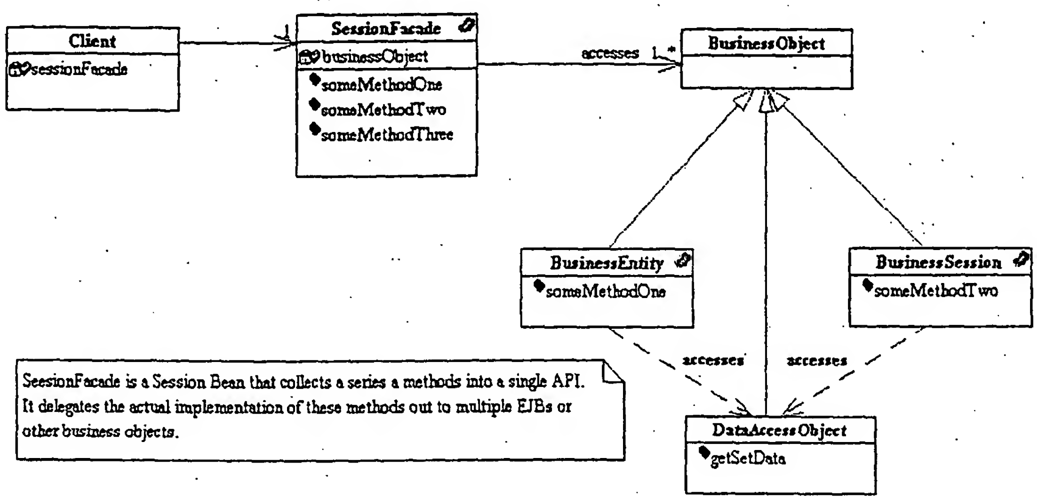
- Photographer – retrieves a photograph from a camera device and stores it to a database or directly onto the file system.
- Locator – records the location of the detection event into a database.
- Movie Player – plays a movie on the computer monitor.

Session Facade Pattern

When several business objects are needed in concert, a single object can create a façade, hiding the complexity of those business objects from the client. A session bean is used as a facade to encapsulate the complexity of interactions between the business objects participating in the workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

Service Activator Pattern

The function of the Service Activator is to receive asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously.



Class Diagrams

Fig. 22

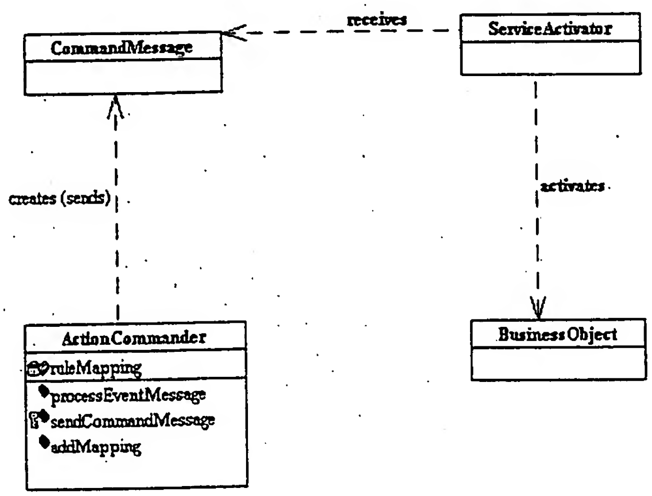


Fig. 23

Sequence Diagrams

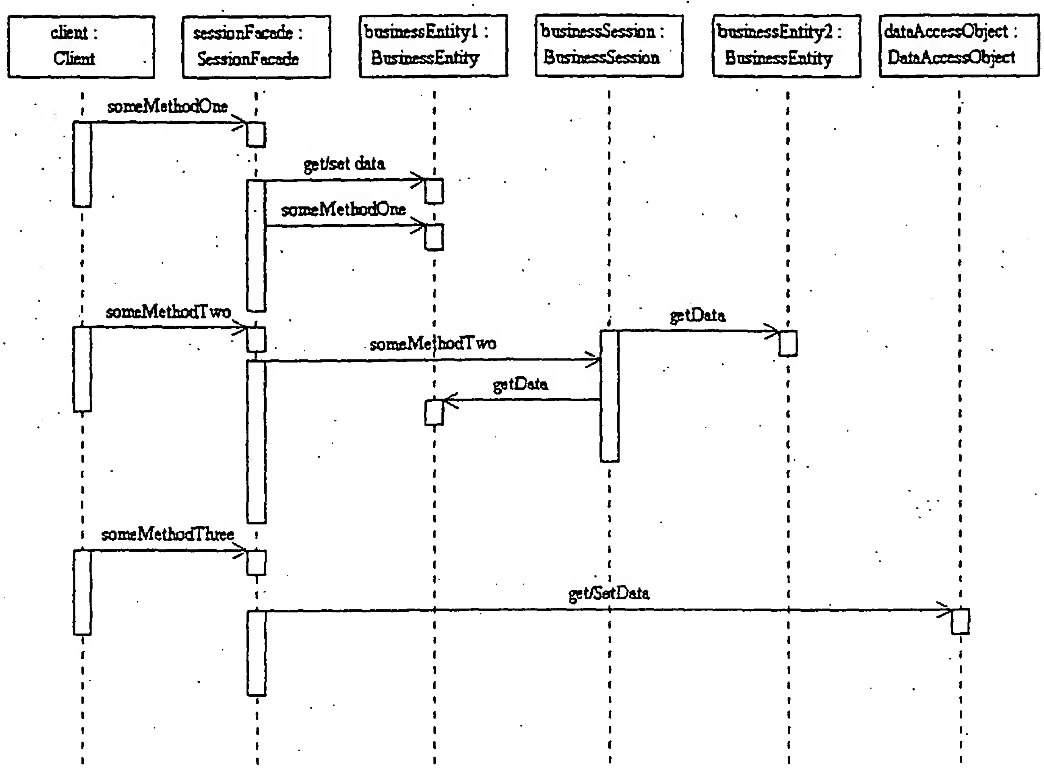

client :
Client
sessionFacade :
SessionFacade
businessEntity1 :
BusinessEntity
businessSession :
BusinessSession
businessEntity2 :
BusinessEntity
dataAccessObject :
DataAccessObject
someMethodOne
get/set data
someMethodOne
someMethodTwo
getData
someMethodTwo
getData
someMethodThree
get/SetData

Fig. 24

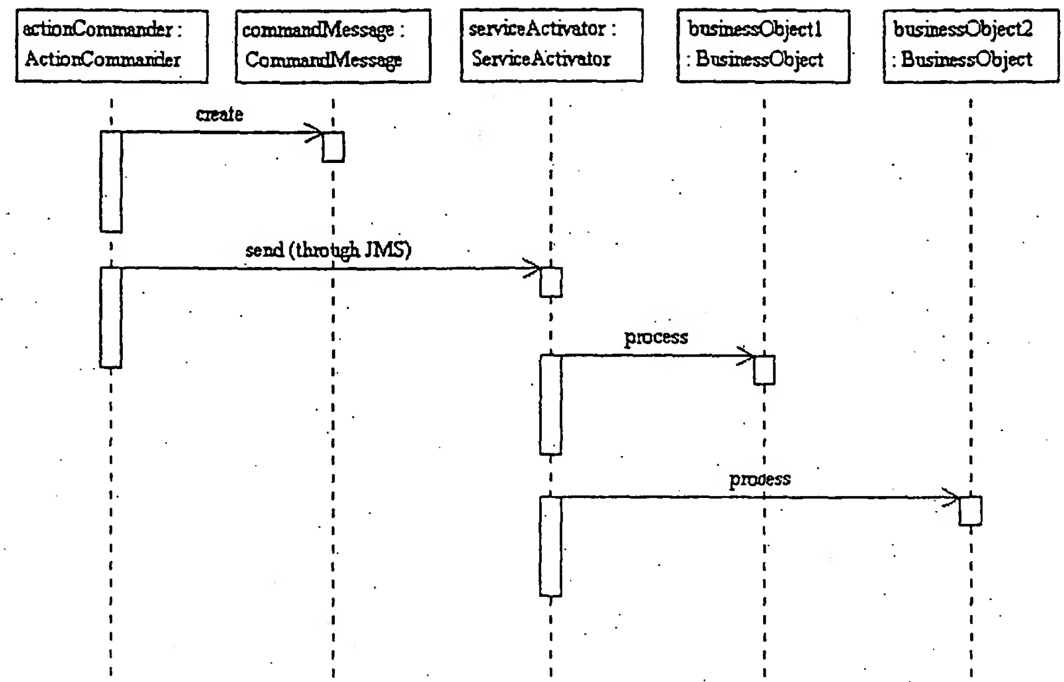

actionCommander : ActionCommander
commandMessage : CommandMessage
serviceActivator : ServiceActivator
businessObject1 : BusinessObject
businessObject2 : BusinessObject
create
send (through JMS)
process
process

Fig. 25

Deployment Options

Because the TRCF is designed to be flexible and scalable, it can be deployed on a wide range of hardware, from an inexpensive PC to a fully redundant, fault tolerant Unix network. Different business needs will require different configurations, so determining the best deployment strategy will require careful assessment of the enterprise and the proposed solution. Instead of attempting to describe every conceivable configuration, this section is focused on a high-end and a low-end solution.

High-End and Low-End Configurations

The following scenarios provide logical deployment options and the strengths and weaknesses of each.

Issue	Low-End System	High-End System
Budget < $20,000?	x	
No existing network infrastructure	x	
Legacy CRM Integration Required		x
Average Load > 200 transactions per minute		x
24/7 operations mandatory		x
Application is mission critical or life dependent		x
Span geographies, such as multiple cities or cross a WAN boundary		x
Potential expansion to meet a much higher load in the future		x
Many complex custom business processes (thick client required)		x
Security in a Web (public Internet) environment required		x
Service agreements from vendors required		x

Low-End Configuration

(1) Combined Web / Application / Database Server

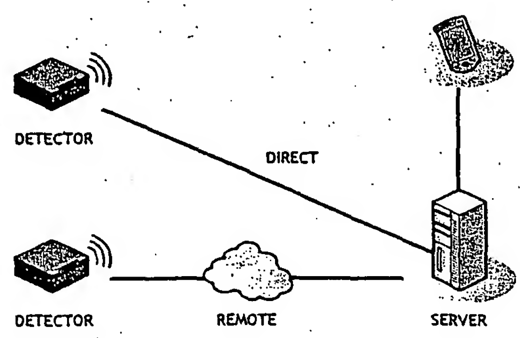


Figure 15 above depicts a low-end configuration. There are no load balancing, clustering, or fail-over strategies provided in this configuration.

Strengths:

- lowest cost
- easiest administration
- available with either software configuration
- still handles many possible business scenarios

Weaknesses:

- not fault tolerant
- lowest throughput capacity

Note that this configuration can be scaled horizontally by adding servers for the application and/or database to gain better performance.

Estimated Peak Load (Detection Events per second):

Price Range	Intel	Sun
Low	20	30
Medium	40	50
High	60	100

High End Configuration

(2) Web Servers w/Load Balancer

(2) Clustered Application Servers

(2) Clustered Database Server

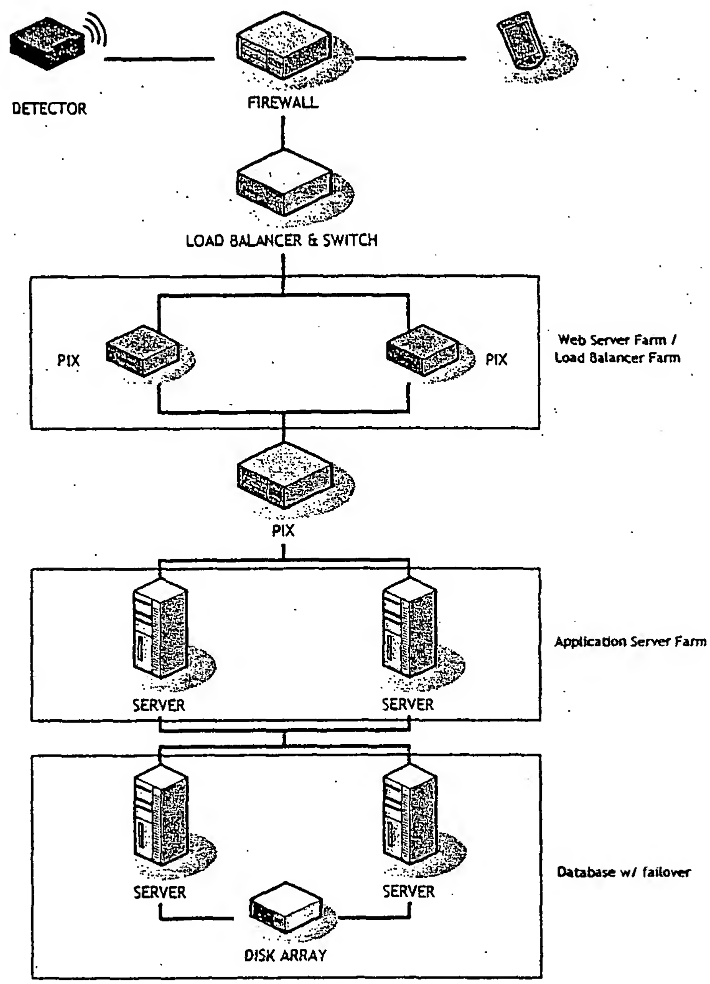


Figure 16 above represents a high-end configuration, designed to handle a large quantity of detection events, complicated business rules that determine actions, and a large, complicated database of information. The system has many fail-over components built in at all three levels of the physical architecture. The dual web servers balance the load, and if one fails, the other will handle the expected load. Clustering is employed at the application server and database server levels of the architecture to ensure load and fail-over handling. This system would be recommended anywhere that uptime is crucial, and a large number of detection events, and action components are deployed. Security is also addressed by using firewalls to protect the

systems from unauthorized access. This may be required in ASP-model deployments, where the detection events come in over the Internet. Even Intranet environments may have need for this configuration, if the system is tracking security-related events.

Strengths:

- Maximum amount of throughput
- Maximum protection from single point machine failures at all three tiers of the system.

Weaknesses:

- Not available with JBoss option.

Note that this configuration can be scaled down to accommodate clients with differing needs.

Estimated Peak Load (Detection Events per second):

Price Range	Intel	Sun
Low	130	200
Medium	170	350
High	200	500

Choosing Software

Choosing the software foundation for a client's deployment configuration requires careful consideration of the business requirements and needs of the particular business processes being supported.

In this section, two software options are presented: JBoss – the "Low End" solution, and WebLogic – the "High End" solution. An application server must be J2EE EJB (Enterprise Java Bean) 2.0 compliant in order to support the basic services required in the TRCF. Message Driven Bean support in particular is required. An application server that is EJB 2.0 compliant supports Message Driven Beans. Our low-end selection, JBoss, is EJB 2.0 compliant, is very low cost and enjoys a good reputation within the J2EE development community. Our high-end solution, BEA WebLogic, is also EJB 2.0 compliant. WebLogic is a leader in the J2EE community. WebLogic includes clustering and fail over. WebLogic can easily scale to support a robust and high volume system. The WebLogic selection includes the possibility of customer support from BEA and training for IT staff. WebLogic also is the supporting infrastructure for other BEA products, such the Business Process Manager, Portal System or Enterprise Integration Server that Tabula Rasa may wish to use when building solutions for its customers.

Following are factors that should be considered when selecting either JBoss or WebLogic:

Certification

J2EE licensing & certification may be a requirement for the choice of an application server.

- JBoss: not licensed or certified
- BEA WebLogic: licensed and certified

Support

Application support consists of documentation, online help, call center support, and mailing lists.

- JBoss: Documentation and mail lists
- BEA WebLogic: All forms of support

Clustering

Clustering allows the administrator to construct a more fault-tolerant system, with load balancing, redundant systems, and fail-over strategies.

- JBoss: does not currently support clustering
- BEA WebLogic: clustering supported

Connectivity

The J2EE Connector Architecture handles connectivity to external systems. Both WebLogic and JBoss provided support for this specification, but in practice, WebLogic has more proven deployments and experience connecting to systems such as SAP, PeopleSoft, Seibel, etc.

Software Options

JBoss-centered solution (Low End)

Software	Product	Price
Web Server	Tomcat v4.0	Free
Application Server	JBoss v2.4.3	Free
Relational Database	Hypersonic v1.6	Free

WebLogic-centered solution (High End)

Software	Product	Price
Web Server	Apache Stronghold	$1000
Application Server	BEA WebLogic v6.1	$10000 per CPU
Relational Database	Oracle	$10000 per CPU for 2 years

Choosing Hardware

Perhaps the most important factor in making the decision about what hardware to buy, is knowing what the client is most experienced and comfortable with. Either software option described above will run on Intel-based hardware, or others such as Sun Sparc-based equipment. Traditionally, BEA WebLogic systems are deployed on higher-end Sparc-based systems due to factors such as operating system robustness, and performance capabilities.

However, low and high end Intel and Sparc based systems are available that would allow vertical scaling of either system to handle most possible loads. Capacity planning is required to determine:

- Processor
- Memory
- Disk
- Network

Capacity factors include anticipated number of transactions, understanding of how different types of transactions are processed, number of targets in the database, amount of data stored about targets, etc.

Hardware Options

Intel

Price Range	Description	Price
Low	Dell PowerEdge 1400SC, Intel® Pentium III 1.26GHz w/512K Cache, 256MB SDRAM, 18GB SCSI, RED HAT LINUX 7.1, Tower Chassis	$1488
Medium	Dell PowerEdge 2500 Intel Pentium III 1.13GHZ w/512K CACHE, 256MB SDRAM, 18GB SCSI, Red Hat Linux 7.1, Tower Chassis	$2237
High	Dell PowerEdge 4400 Dual Processor Intel Pentium III Xeon 1GHz/256K Cache, 512Mb SDRAM, Dual 18GB Ultra3 SCSI 10K RPM Hard Drive, Red Hat Linux 7.1, Tower Chassis	$5338

Sun

Price Range	Description	Price
Low	Sun Enterprise Ultra 5S Server 400Mhz UltraSPARC-IIi Processor, 256 MB RAM, 20GB SCSI, Solaris 8	$2895
Medium	Sun Enterprise 250 Server 400MHZ UltraSPARC-II Processor, 512 MB RAM, 18 GB 10000 RPM UltraSCSI, Solaris 8	$ 7995
High	Sun Enterprise 420R Server Dual 450 MHZ UltraSPARC-II Processor, 2 GB RAM, Dual 18GB 10000 RPM UltraSCSI Disks, Solaris 8	$23795

Any of the above configurations can be used for any of the combined, or separate machine configurations.